

Mailing Label No. EL368756754US

[illegible]

Aufgrund der aber immer noch beschränkten Leistungsfähigkeit solcher Systeme müssen Kompromisse zwischen der Bildqualität und der Echtzeitanforderung (mindestens 25 Bilder pro Sekunde für eine kontinuierliche Bewegung) eingegangen werden. Im Allgemeinen wird mehr Wert auf eine ruckfreie Darstellung gelegt, wodurch einerseits Objekte nur sehr grob modelliert werden, um die Anzahl der Polygone klein zu halten, und andererseits die Bildschirmauflösung klein gehalten wird, um die Anzahl der zu generierenden Pixel zu begrenzen. Bei heute üblicher VESA-Auflösung (640 x 480 Pixel) und animierten Bildsequenzen fallen Aliasing-Effekte, die durch die Rasterisierung entstehen, aber noch besonders störend ins Auge. Trotzdem werden sie toleriert, da klassische Antialiasing-Ansätze, wie zum Beispiel das Supersampling, einen zu hohen Speicher- und Rechenleistungsbedarf haben.

Bis eine im Computer modellierte Szene auf dem Bildschirm dargestellt werden kann, bedarf es mehrerer Schritte:

1. Die im Speicher abgelegten Datensätze der darzustellenden Objekte müssen transformiert (skaliert, rotiert) und an den richtigen Stellen in der virtuellen Szene platziert werden (modeling transformation).
2. Ausgehend von der Position der Objekte bezüglich des Blickwinkels des Betrachters werden nun Objekte verworfen und damit nicht weiterberücksichtigt, die garantiert nicht sichtbar sein können. Dabei werden sowohl ganze Objekte eliminiert, die sich außerhalb des sichtbaren Volumens befinden (clipping), als auch einzelne Polygone von Objekten, die dem Betrachter abgewendet sind (backface removal).
3. Die in Weltkoordinaten modellierten Polygone (meist Dreiecke) müssen nun in das Bildkoordinatensystem überführt werden, wobei eine perspektivische Verzerrung erfolgt, um eine möglichst realistische Abbildung zu ermöglichen (viewing transformation).
4. Die nun in Bildkoordinaten vorliegenden Polygone müssen so aufbereitet werden, daß sie durch den Renderer verarbeitet werden können (z.B. Berechnung von Steigungen an Kanten, usw. / setup processing).
5. Im Rasterisierer erfolgt dann eine Berechnung der sichtbaren Pixel auf dem Bildschirm. Pro Pixel wird nicht nur die Position auf dem Bildschirm (x, y)

berechnet, sondern es erfolgt auch die Bestimmung weiterer für die Beleuchtung und Verdecktheitsanalyse benötigter Parameter (z-Wert, homogener Parameter, Texturkoordinaten, Normalen, usw. / rasterization)

6. Aufgrund der berechneten Parameter werden nun die Farbwerte der darzustellenden Pixel ermittelt (lighting). Dieser Schritt wird hier nur dann durchgeführt, wenn es sich um ein Phong-Renderer handelt. Ist nur ein Gouraud-Renderer vorhanden, so wird dieser Schritt bereits vor der Transformation ins Bildkoordinatensystem durchgeführt.
7. Die berechneten Farbwerte werden dann im Framebuffer gespeichert, wenn der z-Wert des Pixels angibt, daß das Pixel sich vor dem an dieser Position im Framebuffer stehendem Pixel befindet (z-buffering). Vor der Speicherung können die Farbwerte mittels des Blendings mit dem vorher im Framebuffer stehen Wert modifiziert werden, wodurch z.B. die Modellierung von halb transparenten Objekten möglich wird.
8. Sind alle sichtbaren Dreiecke rasterisiert worden, so befindet sich im Framebuffer das auf dem Bildschirm darzustellende Bild. Über die RAMDAC wird das Bild linear aus dem Speicher ausgelesen und in analoge Signale gewandelt, die direkt an den Monitor geschickt werden (display process).

Das Problem des Aliasing entsteht aus der heute üblichen Verwendung von Rasterdisplays, da es mit diskreten Punkten nicht möglich ist, beispielsweise eine geneigte Kante exakt darzustellen. Bei der "normalen" Scankonvertierung wird ein Pixel (picture element) immer dann gesetzt, wenn der Pixelmittelpunkt bedeckt ist, wodurch aus einer kontinuierlichen Kante im diskreten Fall in bestimmten Abständen sichtbare Sprünge entstehen, die bei bewegten Bildern besonders auffallen, da sie über die Kante wandern. So tritt zum Beispiel bei der Bewegung einer fast horizontalen Kante der irritierende Effekt auf, daß, wenn die Kante langsam vertikal verschoben wird, die Sprünge schnell horizontal über die Kante laufen. Es scheint also weder die Bewegungsrichtung noch die Bewegungsgeschwindigkeit zu stimmen.

Aus der Signalverarbeitung ist bekannt, daß sich ein Signal (in diesem Fall das Bild) nur dann korrekt reproduzieren läßt, wenn die Abtastrate größer als das Doppelte der maximal auftretenden Frequenz ist (Abtasttheorem von Shannon). Da aber die Abtastrate durch die Bildschirmauflösung fest vorgegeben ist, werden als

Das Problem des klassischen Samplings besteht also darin, daß ein Pixel immer als Punkt betrachtet wird, wodurch sich auch der Name Point-Sampling ergibt. Allen Antialiasing-Ansätzen gemein ist, daß das Pixel nun als etwas Flächenhaftes gesehen wird; die Farbe sich ergo aus der Mittelung der Farbgebung der bedeckten Pixelfläche ergeben sollte. Das Antialiasing versucht nun die darstellungsbedingten Probleme soweit wie möglich zu beseitigen, oder zumindest abzuschwächen.

Sehr kleine Objekte können verschwinden, falls ihre Ausdehnung kleiner als ein Pixel ist. Dieser Effekt fällt besonders dann auf, wenn durch eine kleine Verschiebung das Objekt sichtbar wird, da auf einmal ein Pixelmittelpunkt berührt wird. Es entsteht eine Art Blinkereffekt, der die Aufmerksamkeit des Betrachters auf sich lenkt.

Der am weitesten verbreitete Ansatz für das Antialiasing ist das Supersampling. Jedes Pixel wird in $(n \times n)$ Subpixel unterteilt, die dann wiederum normal "gepointsampelt" werden. Es ergibt sich ein Zwischenbild, welches in beiden Dimensionen die n -fache Auflösung des darzustellenden Bildes besitzt. Durch Aufsummierung der Farbwerte der $(n \times n)$ Subpixel und der anschließende Division durch die Anzahl der Subpixel (n^2) ergibt sich dann die endgültige Farbe des Pixels. Aus Sicht der Signalverarbeitung wird die Abtastrate um den Faktor n (auch Oversamplingfaktor genannt) erhöht, wodurch kleinere Details rekonstruiert werden können. Sinnvolle Werte für n liegen im Bereich 2 bis 8, wodurch zwischen 4 und 64 Farbwerte pro Pixel zur Verfügung stehen.

Trotz der Einfachheit dieses Vorgehens gibt es entscheidende Nachteile, aufgrund derer eine Realisierung in Hardware nicht erreicht werden konnte:

1. Speicherbedarf: Da das Bild in einer n -fachen Auflösung gerendert wird, muß nicht nur der Framebuffer, sondern auch der z-Buffer (24 - 32Bit/Pixel) n -fach ausgelegt sein.

Beispielsweise ergibt sich für eine Bildschirmauflösung von 1024 x 768 Pixel und einen Oversamplingfaktor von $n=4$ ein Speicherbedarf von 48 MB für den Framebuffer und 36 MB noch einmal für den z-Buffer. Insgesamt werden also 84 MByte Speicher gegenüber 5,25 MB für das normale Sampling benötigt.

2. Rechenzeitaufwand: Aufgrund der höheren Anzahl der zu erzeugenden Pixel nimmt die Rechenzeit ebenfalls um den Faktor n^2 zu. War das System also vorher in der Lage, 16 Bilder pro Sekunde anzuzeigen, so kommt es bei $n=4$ nur noch auf ein Bild pro Sekunde; es hat seine Echtzeitfähigkeit also verloren.

Das Verfahren kann zudem nicht garantieren, daß das entstehende Bild frei von Artefakten ist, denn zu jeder Samplingrate läßt sich leicht ein Bild konstruieren, welches garantiert falsch dargestellt wird. Hat das darzustellende Bild eine horizontale Auflösung von w , so wird ein senkrechtes Streifenmuster aus $(n \times w)$ schwarzen und $(n \times w)$ weißen Balken, entweder komplett schwarz oder komplett weiß dargestellt.

Beim stochastischen Supersampling werden die Samplepoints zufällig über das Pixel verteilt, wodurch die übrig gebliebenen Artefakte mit einem Rauschen überlagert werden, welches für das menschliche Auge angenehmer ist.

Ein Bild, welches vierfach mit einem stochastischen Ansatz gerendert wurde, hat in etwa die gleiche Bildqualität, wie ein 16faches Supersampling an einem regulären Raster.

Das Verfahren ist jedoch auf eine Anwendung in Software beschränkt, da Hardware-Renderer ausschließlich mit inkrementellen Verfahren arbeiten. Bei beliebig platzierten Samplepoints gibt es keine feste Reihenfolge der Punkte mehr, so daß die Bearbeitung nicht mehr inkrementell erfolgen kann, sondern die Parameter pro Punkt komplett neu berechnet werden müßten, was einen extremen Aufwand bedeutet.

Supersampling erfolgt in seiner Grundform über das gesamte Bild immer mit der gleichen Abtastrate. In flächigen Bereichen ist jedoch viel Rechenzeit erforderlich, da dort jedes Subpixel den gleichen Farbwert beisteuert. Die Idee besteht dabei darin, Supersampling nur dort zu betreiben, wo es wirklich nötig ist. In einem ersten Durchgang wird das Bild normal gerendert, und in einem zweiten Durchlauf wird dann jeder Farbwert mit den Farbwerten aus seiner Umgebung verglichen, und nur sofern die Differenz einen vorgegebener Schwellwert übersteigt, erfolgt nun ein Supersampling für dieses Pixel. Der Nachteil bei diesem Vorgehen ist natürlich das zweimalige Rendern eines jeden Polygons.

Der Accumulationbuffer ist eine Abwandlung des vorgenannten Supersamplings, bei dem der extreme Speicherbedarf vermieden wird. Es wird lediglich zu dem sowieso vorhandenen Frame- bzw. z-Buffer ein zusätzlicher Buffer von der Größe des Framebuffers benötigt, der jedoch eine etwas höhere Genauigkeit braucht. Das Rendern eines Bildes benötigt nun n^2 Renderdurchgänge bei der normalen Framebufferauflösung, wobei n wiederum den Oversamplingfaktor darstellt. Zwischen den Berechnungen der Teilbilder wird jeweils das Koordinatensystem der zu beschreibenden Geometrie im Subpixelbereich so verschoben, daß die Pixelmittelpunkte jeweils auf einem anderen Sample des entsprechenden Supersampling-Verfahrens zu liegen kommen. Die bei jedem Rendering-Durchlauf erzeugten Framebuffer-Inhalte werden im zusätzlichen Buffer akkumuliert (daher der Name Accumulationbuffer), worauf sich das Löschen des Frame- und z-Buffer für das nächste Teilbild anschließt. Da im Accumulationbuffer n^2 Farbwerte pro Pixel aufsummiert werden, sind zu der Framebuffergenauigkeit noch einmal $2 \cdot \log_2 n$ Genauigkeitsbits zusätzlich erforderlich, damit kein Überlaufen der Farbwerte möglich ist. Sobald alle Teilbilder gerendert wurden, werden die Farbwerte aus dem Accumulationbuffer durch die Anzahl der Samples (n^2) geteilt, und in den Framebuffer übernommen, der dann angezeigt werden kann.

Durch den Einsatz des Accumulationbuffers anstelle des beim Supersamplings riesigen Framebuffers wird der Nachteil des großen Speicherbedarfs beseitigt, nicht jedoch der Bedarf an Rechenzeit. Im Gegenteil wird sich die Renderingzeit sogar

Das Areasampling, entwickelt von Edwin Catmull [Edwin Catmull: "A hidden-surface algorithm with antialiasing", Aug.78], basiert darauf, daß pro Pixel die Fläche berechnet wird, welche auf die einzelnen Polygone entfällt. Dies geschieht auf analytischem Wege, so daß jedes noch so kleine Polygon in Betracht gezogen wird, und liefert eine dementsprechend hohe Bildqualität.

- Alle Polygone werden nach ihrer größten y-Koordinate geordnet.
- Es wird eine Liste von aktiven Polygonen verwaltet, in die Polygone eingetragen werden, sobald die Scanline mit dem größtem y-Wert erreicht ist, und aus der sie wieder gelöscht werden, sobald der minimale y-Wert unterschritten wird.
- Pro Scanline, wobei eine Scanline hierbei als etwas Flächenhaftes betrachtet wird, muß pro Pixel ein Bucket angelegt werden, in dem jeweils alle Polygonanteile eingetragen werden, die zu diesem Pixel beitragen. Die Polygonanteile werden dabei wieder durch Polygone repräsentiert, die gegenüber den Pixelkanten geclippt wurden. Beim Aufbau der Buckets wird darauf geachtet, daß die jeweiligen Polygone entsprechend ihrer z-Werte einsortiert werden.
- Pro Pixel wird nun mit Hilfe des sogenannten "Hidden-Surface-Algorithmus" von Sutherland die sichtbare Fläche der einzelnen Polygone bestimmt, deren Farben dann gewichtet die endgültige Pixelfarbe ergeben.

Der A-Buffer-Algorithmus [Loren Carpenter: "The a-buffer, an antialiased hidden surface method", Computer Graphics 12, 84]) stellt die diskrete Realisierung des Areasamplings dar, bei der nicht die exakte Fläche der Polygone abgespeichert wird, sondern nur Subpixelmasken, die eine Approximation der Flächen darstellen. Zudem erfolgt die Traversierung polygonweise, so daß wiederum ein Framebuffer benötigt wird, der jedoch eine dynamische Liste pro Pixel aufnehmen muß, in der

Der konstant hohe Speicherbedarf des Supersamplings wurde somit durch einen sich dynamisch der Komplexität der Szene anpassenden Speicherbedarf ersetzt. Aufgrund dieses dynamischen Verhaltens und der daraus resultierenden Traversierung von Listen eignet sich dieses Verfahren kaum für eine Hardware-Implementierung. Weiterhin stellt sich die Verdecktheitsanalyse anhand der begrenzten Anzahl der z-Werte als problematisch heraus. In [Andreas Schilling und Wolfgang Straßer: "EXACT: Algorithm and Hardware Architecture for an Improved A-Buffer", Computer Graphics Proceedings, 93] wird eine mögliche Lösung mittels der zusätzlichen Speicherung von dz/dx und dz/dy auf Subpixelebene aufgezeigt.

Beim Approximationbuffer von Lau [Wing Hung Lau: "The antialiased approximation buffer", Aug.94] findet ebenfalls der A-Buffer-Ansatz Verwendung, wobei jedoch die Anzahl der pro Pixel gespeicherten Fragmente auf zwei beschränkt wird. Es ergibt sich somit ein konstanter Speicheraufwand, der jedoch mit Verlusten in der Bildqualität erkauft wird. So werden nur noch Pixel korrekt behandelt, die maximal von zwei Polygonen bedeckt sind, da mehr Anteile nicht darstellbar sind. Das Verfahren ist demnach auf wenige, große Polygone beschränkt, da dann der Fall von mehr als zwei Fragmenten praktisch nur noch sehr selten auftritt (unter 0,8% laut Lau), und somit die Qualität der Ergebnisbilder ausreichend gut ist.

Der Exact Area Subpixel Algorithm (EASA von Andreas Schilling [Andreas Schilling: "Eine Prozessor-Pipeline zur Anwendung in der graphischen Datenverarbeitung", Dissertation an der Eberhard-Karls-Universität in Tübingen, Juni 94]) stellt eine Modifikation des A-Buffers dar, bei dem eine höhere Genauigkeit an Kanten eines Polygons erreicht wird. Die Generierung der Subpixelmasken erfolgt beim A-Buffer aufgrund der Bedeckung der Subpixelmittelpunkte. Schilling hingegen berechnet den exakten Flächenanteil, woraus eine Anzahl zu setzender Subpixel abgeleitet wird, die dann zur Generierung der eigentlichen Maske anhand der Steigung der Kante führt. Durch dieses Vorgehen kann eine höhere Auflösung bei fast horizontalen (vertikalen) Kanten erreicht werden, da beim A-Buffer immer mehrere Subpixel auf einmal gesetzt werden, so daß nicht die maximale Auflösung der Subpixelmaske

ausgenutzt werden konnte.

Das Verfahren von Patrick Baudisch [Patrick Baudisch: "Entwicklung und Implementierung eines effizienten, hardwarenahen Antialiasing-Algorithmus", Diplomarbeit, Technische Hochschule Darmstadt, Sept.94]) basiert ebenfalls auf Subpixelmasken, die an Polygonkanten generiert werden. Jedoch dienen sie hierbei nicht dazu, die Fläche der einzelnen Polygone zu berechnen und deren Farbe damit entsprechend zu gewichten wie bei den vorigen Verfahren, sondern um aus einem normal gepointsampteten Bild die benachbarten Farben zuzumischen. Als Grundlage dient die räumliche Kohärenz der Pixel, d.h., daß die Farbe, die ein Polygon teilweise zu einem Pixel beiträgt, garantiert in einem benachbarten Pixel zu finden ist. Die Position der Subpixel in der Maske gibt an, aus welchem benachbartem Pixel eine Zumischung erfolgen soll, sofern es gesetzt wird. Die Subpixel auf den Diagonalen verweisen dabei auf zwei benachbarte Pixel.

Beim Vier-Zeiger-Verfahren werden jeweils 4 Subpixel zu einem Meta-Subpixel zusammengefaßt, welches wiederum die Zumischung der benachbarten Pixelfarbe angibt. Durch das Zusammenfassen geht die räumliche Information verloren, die Genauigkeit der Flächenanteile erhöht sich aber. Die bisherigen Antialiasing-Verfahren haben insgesamt Nachteile in bezug auf ihre mögliche Hardware-Realisierbarkeit, die nur schwer behoben werden können. Einerseits muß ein enormer Speicheraufwand betrieben werden (Supersampling, Area-sampling, A-Buffer), und andererseits ist der Rechenaufwand einiger Verfahren so hoch (Supersampling, Accumulationbuffer, Areasampling), daß eine Hardware-Realisierung kaum noch echtzeitfähig ist. Zudem werden bei Verfahren, die ausschließlich auf Polygonkanten arbeiten, (Areasampling, A-Buffer, Approximationbuffer, EASA, Subpixelverfahren, Vier-Zeiger-Verfahren) Billboard- und Spot-Artefakte nicht berücksichtigt.

Die beim Stand der Technik bestehenden Probleme sollen nachfolgend noch einmal kurz zusammengefaßt werden:

Das Problem des Aliasing entsteht aus der Verwendung von Rasterdisplays, da es mit diskreten Punkten nicht möglich ist, z.B. eine geneigte Kante exakt darzustellen. Bei normalen Rasterisierungsmethoden entstehen aus einer kontinuierlichen Kante im diskreten Fall in bestimmten Abständen Sprünge, die den visuellen Eindruck stark stören.

Ein üblicher Ansatz um diesen Effekt zu beheben, ist das Supersampling, bei dem durch viele Abtastpunkte innerhalb eines Pixels versucht wird, eine bessere Farbe zu bestimmen. Das Verfahren ist aber kaum im Echtzeitbereich einsetzbar, da ein sehr hoher Speicher- und Rechenzeitaufwand nötig ist.

Die anderen Verfahren versuchen, einen besseren Farbwert zu bestimmen, indem sie den flächenmäßigen Beitrag der Polygone zu jedem Pixel genau berechnen. Aber auch hier ist ein sehr hoher Rechenzeitbedarf (wenn nicht auch Speicherplatz) notwendig.

Aus der US-Patentschrift 5 748 178 ist es ferner bekannt, die Umgebung eines Pixels im Durchlauf in einer Art Schieberegister zu speichern, wobei einander benachbarte Pixel auch benachbarte Speicherplätze einnehmen. Eine Filterung wird dabei dadurch erzielt, daß dann jeweils eine Pixel-Umgebung einem gemeinsamen Filter- Gewichtungsvorgang unterworfen werden kann. Da die Wirksamkeit des Verfahrens davon abhängig ist, welche Pixel einander zufällig benachbart sind, ist hiermit ein wirksames Antialiasing nicht möglich.

Aus der US-Patentschrift 5 264 838 ist es ebenfalls bekannt, zum Zwecke des Antialiasing je eine Umgebung eines Pixels im Bereich eines Impulses mit einer unscharfen Umgebung (Halo) zu versehen. Dieses Verfahren erzeugt jedoch lediglich eine zusätzliche Unschärfe, da es ungezielt wirkt.

Der Erfindung liegt demgegenüber die Aufgabe zugrunde, ein Antialiasing-Verfahren anzugeben, welches die Echtzeitfähigkeit nicht beeinträchtigt, wodurch trotz der Notwendigkeit der Realisierung in Hardware eine deutliche Verbesserung der Bildqualität erreicht werden konnte.

Diese Aufgabe wird durch ein Verfahren gelöst, welches zum Eliminieren unerwünschter Stufungen an Kanten bei Bilddarstellungen im Zeilenraster, insbesondere im On-line Betrieb dient. Dieses Verfahren ist gekennzeichnet durch die Schritte:

- a) Anwendung eines Kantenoperators auf einen Bildteil zur Grobermittlung mindestens eines gerasterten Kantenverlaufs,
- b) Bestimmung der Position mindestens eines ersten Pixels aus der Menge derjenigen Pixel, die den gerasterten Kantenverlauf bilden oder

an diesem gerasterten Kantenverlauf angrenzen,

- c) Approximation einer Geraden zur Ermittlung eines wahrscheinlichen Verlaufs der ungerasterten Bildkante in der Nähe des ersten Pixels,
- d) Ermittlung eines Kriteriums aus der Approximationsgeraden und der Position des ersten Pixels für die Zumischung einer Farbe X zu der Farbe C im betrachteten ersten Pixels und
- e) Mischung der ermittelten Farbe X zu der Farbe C im betrachteten ersten Pixel.

Gemäß einer besonders bevorzugten Ausführungsform ist das erfindungsgemäße Verfahren dadurch gekennzeichnet, daß das Kriterium des Verfahrensschrittes d) in Abhängigkeit von der Lage des betrachteten Pixels relativ zur Approximationsgeraden festlegt, welche Farbe X zu der Farbe C des betrachteten Pixels zugemischt wird.

Gemäß einer besonders bevorzugten Ausführungsform ist das erfindungsgemäße Verfahren dadurch gekennzeichnet, daß das Kriterium gemäß Verfahrensschritt d) in Abhängigkeit von der Lage des betrachteten Pixels relativ zur Approximationsgeraden festlegt, daß die Farbe mindestens eines benachbarten Pixels gewichtet zur Farbe des betrachteten Pixels zugemischt wird.

Gemäß einer besonders bevorzugten Ausführungsform ist das erfindungsgemäße Verfahren dadurch gekennzeichnet, daß bei einem betrachteten Pixel, das von der Approximationsgeraden nicht geschnitten wird, die Farbe unverändert bleibt.

Gemäß einer besonders bevorzugten Ausführungsform ist das erfindungsgemäße Verfahren dadurch gekennzeichnet, daß bei einem betrachteten Pixel, das von der Approximationsgeraden geschnitten wird, die resultierende Farbe R nach Maßgabe des folgenden Kriteriums bestimmt wird:

die Approximationsgerade teilt das betrachtete Pixel in zwei Teilflächen F_1 , F_2 , wobei $F_1 + F_2 = 1$, mit 1 ist die Gesamtfläche des Pixels, wobei F_1 diejenige Teilfläche ist, in welcher der Pixelmittelpunkt liegt;

- zugemischt wird zu der Farbe C des betrachteten Pixels die Farbe X

Gemäß einer besonders bevorzugten Ausführungsform ist das erfindungsgemäße Verfahren dadurch gekennzeichnet, daß die Approximationsgerade über mehrere Stufen des gerasterten Kantenverlaufs verläuft, und daß die Approximationsgerade

endet, wenn die Kriterien

- 1) Es können maximal zwei verschiedene Stufenlängen vorkommen, deren Stufenlängen sich außerdem um maximal 1 unterscheiden dürfen.
- 2) Nur eine der beiden Stufenlängen darf mehrmals hintereinander auftreten.
- 3) Durch das Aneinanderreihen der Anzahlen der Stufen, die die gleiche Länge haben, erhält man eine Zahlensequenz, bei der abwechselnd immer eine Eins und dann eine beliebige Zahl (> 0) steht. Die Einsen (nur die an jeder zweiten Position) werden aus dieser Sequenz gestrichen. Bei der erhaltenen Sequenz dürfen wieder nur zwei verschiedene Zahlen vorkommen, die sich um eins unterscheiden.
- 4) Bei der unter 3. erhaltenen Sequenz darf nur eine der beiden möglichen Zahlen mehrmals hintereinander auftreten.
- 5) Durch wiederholtes Anwenden der Regeln 3. und 4. auf die Zahlensequenz, läßt sich ein immer globalerer Blick auf die Kante gewinnen.

in aufsteigender Reihenfolge überprüft werden und mindestens ein Kriterium nicht erfüllt ist.

Gemäß einer besonders bevorzugten Ausführungsform ist das erfindungsgemäße Verfahren dadurch gekennzeichnet, daß die Approximationsgerade über mehrere Stufen des gerasterten Kantenverlaufs verläuft, und daß die Approximationsgerade endet, wenn eines der Kriterien

- 1) Es können maximal zwei verschiedene Stufenlängen vorkommen, deren Stufenlängen sich außerdem um maximal 1 unterscheiden dürfen.
- 2) Nur eine der beiden Stufenlängen darf mehrmals hintereinander auftreten.
- 3) Durch das Aneinanderreihen der Anzahl der Stufen, die die gleiche

4) Bei der unter 3. erhaltenen Sequenz darf nur eine der beiden möglichen Zahlen mehrmals hintereinander auftreten.

... einer besonders bevorzugten Ausführungsform ist das erfindungsgemäße Verfahren gekennzeichnet durch das Vorsehen eines Tripple-Buffers, wobei sich die arbeitenden Buffer in zyklischer Vertauschung parallel die Verfahrensschritte Bildaufnahme, Post-Antialysing und Bildwiedergabe teilen.

Bei der Rasterisierung eines Dreiecks unter Verwendung der normalen Scankonvertierung, wird ein Pixel immer dann in der Farbe des Dreiecks gefärbt, wenn der Pixelmittelpunkt innerhalb der Dreiecksfläche liegt, anderenfalls erhält das Pixel die Hintergrundfarbe. Hierbei entstehen an den Kanten des Dreiecks sichtbare Sprünge, die den visuellen Eindruck einer geraden Kante stark stören. Nach dem erfindungsgemäßen Verfahren wird nun den Pixeln an den Kanten des Dreiecks eine gemischte Farbe gegeben, welche zwischen der Farbe des Dreiecks und der Hintergrundfarbe liegt. Hierbei wird die flächenmäßigen Bedeckung des Pixels als Kriterium herangezogen. Durch dieses Vorgehen lassen sich alle ungewollten Stufeneffekte vermeiden, wodurch die Qualität des Bildes merklich heraufgesetzt wird.

Für die Mischung der Farben wird also die bedeckte Fläche der einzelnen Farbanteile benötigt, die bei herkömmlichen Antialiasing-Verfahren während der Rasterisierung bestimmt wird. Die Information, welche Flächenanteile auf die einzelnen Seiten der Kante entfallen, läßt sich auch aus einem normal gerendertem Bild im nachhinein

1. *Chlorophyll a* (Chl a) 680 nm
 2. *Chlorophyll b* (Chl b) 645 nm
 3. *Chlorophyll c* (Chl c) 620 nm
 4. *Chlorophyll d* (Chl d) 685 nm
 5. *Chlorophyll e* (Chl e) 670 nm
 6. *Chlorophyll f* (Chl f) 680 nm
 7. *Chlorophyll g* (Chl g) 680 nm
 8. *Chlorophyll h* (Chl h) 680 nm
 9. *Chlorophyll i* (Chl i) 680 nm
 10. *Chlorophyll j* (Chl j) 680 nm
 11. *Chlorophyll k* (Chl k) 680 nm
 12. *Chlorophyll l* (Chl l) 680 nm
 13. *Chlorophyll m* (Chl m) 680 nm
 14. *Chlorophyll n* (Chl n) 680 nm
 15. *Chlorophyll o* (Chl o) 680 nm
 16. *Chlorophyll p* (Chl p) 680 nm
 17. *Chlorophyll q* (Chl q) 680 nm
 18. *Chlorophyll r* (Chl r) 680 nm
 19. *Chlorophyll s* (Chl s) 680 nm
 20. *Chlorophyll t* (Chl t) 680 nm
 21. *Chlorophyll u* (Chl u) 680 nm
 22. *Chlorophyll v* (Chl v) 680 nm
 23. *Chlorophyll w* (Chl w) 680 nm
 24. *Chlorophyll x* (Chl x) 680 nm
 25. *Chlorophyll y* (Chl y) 680 nm
 26. *Chlorophyll z* (Chl z) 680 nm
 27. *Chlorophyll aa* (Chl aa) 680 nm
 28. *Chlorophyll ab* (Chl ab) 680 nm
 29. *Chlorophyll ac* (Chl ac) 680 nm
 30. *Chlorophyll ad* (Chl ad) 680 nm
 31. *Chlorophyll ae* (Chl ae) 680 nm
 32. *Chlorophyll af* (Chl af) 680 nm
 33. *Chlorophyll ag* (Chl ag) 680 nm
 34. *Chlorophyll ah* (Chl ah) 680 nm
 35. *Chlorophyll ai* (Chl ai) 680 nm
 36. *Chlorophyll aj* (Chl aj) 680 nm
 37. *Chlorophyll ak* (Chl ak) 680 nm
 38. *Chlorophyll al* (Chl al) 680 nm
 39. *Chlorophyll am* (Chl am) 680 nm
 40. *Chlorophyll an* (Chl an) 680 nm
 41. *Chlorophyll ao* (Chl ao) 680 nm
 42. *Chlorophyll ap* (Chl ap) 680 nm
 43. *Chlorophyll aq* (Chl aq) 680 nm
 44. *Chlorophyll ar* (Chl ar) 680 nm
 45. *Chlorophyll as* (Chl as) 680 nm
 46. *Chlorophyll at* (Chl at) 680 nm
 47. *Chlorophyll au* (Chl au) 680 nm
 48. *Chlorophyll av* (Chl av) 680 nm
 49. *Chlorophyll aw* (Chl aw) 680 nm
 50. *Chlorophyll ax* (Chl ax) 680 nm
 51. *Chlorophyll ay* (Chl ay) 680 nm
 52. *Chlorophyll az* (Chl az) 680 nm
 53. *Chlorophyll ba* (Chl ba) 680 nm
 54. *Chlorophyll bb* (Chl bb) 680 nm
 55. *Chlorophyll bc* (Chl bc) 680 nm
 56. *Chlorophyll bd* (Chl bd) 680 nm
 57. *Chlorophyll be* (Chl be) 680 nm
 58. *Chlorophyll bf* (Chl bf) 680 nm
 59. *Chlorophyll bg* (Chl bg) 680 nm
 60. *Chlorophyll bh* (Chl bh) 680 nm
 61. *Chlorophyll bi* (Chl bi) 680 nm
 62. *Chlorophyll bj* (Chl bj) 680 nm
 63. *Chlorophyll bk* (Chl bk) 680 nm
 64. *Chlorophyll bl* (Chl bl) 680 nm
 65. *Chlorophyll bm* (Chl bm) 680 nm
 66. *Chlorophyll bn* (Chl bn) 680 nm
 67. *Chlorophyll bo* (Chl bo) 680 nm
 68. *Chlorophyll bp* (Chl bp) 680 nm
 69. *Chlorophyll bq* (Chl bq) 680 nm
 70. *Chlorophyll br* (Chl br) 680 nm
 71. *Chlorophyll bs* (Chl bs) 680 nm
 72. *Chlorophyll bt* (Chl bt) 680 nm
 73. *Chlorophyll bu* (Chl bu) 680 nm
 74. *Chlorophyll bv* (Chl bv) 680 nm
 75. *Chlorophyll bw* (Chl bw) 680 nm
 76. *Chlorophyll bx* (Chl bx) 680 nm
 77. *Chlorophyll by* (Chl by) 680 nm
 78. *Chlorophyll bz* (Chl bz) 680 nm
 79. *Chlorophyll ca* (Chl ca) 680 nm
 80. *Chlorophyll cb* (Chl cb) 680 nm
 81. *Chlorophyll cc* (Chl cc) 680 nm
 82. *Chlorophyll cd* (Chl cd) 680 nm
 83. *Chlorophyll ce* (Chl ce) 680 nm
 84. *Chlorophyll cf* (Chl cf) 680 nm
 85. *Chlorophyll cg* (Chl cg) 680 nm
 86. *Chlorophyll ch* (Chl ch) 680 nm
 87. *Chlorophyll ci* (Chl ci) 680 nm
 88. *Chlorophyll cj* (Chl cj) 680 nm
 89. *Chlorophyll ck* (Chl ck) 680 nm
 90. *Chlorophyll cl* (Chl cl) 680 nm
 91. *Chlorophyll cm* (Chl cm) 680 nm
 92. *Chlorophyll cn* (Chl cn) 680 nm
 93. *Chlorophyll co* (Chl co) 680 nm
 94. *Chlorophyll cp* (Chl cp) 680 nm
 95. *Chlorophyll cq* (Chl cq) 680 nm
 96. *Chlorophyll cr* (Chl cr) 680 nm
 97. *Chlorophyll cs* (Chl cs) 680 nm
 98. *Chlorophyll ct* (Chl ct) 680 nm
 99. *Chlorophyll cu* (Chl cu) 680 nm
 100. *Chlorophyll cv* (Chl cv) 680 nm
 101. *Chlorophyll cw* (Chl cw) 680 nm
 102. *Chlorophyll cx* (Chl cx) 680 nm
 103. *Chlorophyll cy* (Chl cy) 680 nm
 104. *Chlorophyll cz* (Chl cz) 680 nm
 105. *Chlorophyll da* (Chl da) 680 nm
 106. *Chlorophyll db* (Chl db) 680 nm
 107. *Chlorophyll dc* (Chl dc) 680 nm
 108. *Chlorophyll dd* (Chl dd) 680 nm
 109. *Chlorophyll de* (Chl de) 680 nm
 110. *Chlorophyll df* (Chl df) 680 nm
 111. *Chlorophyll dg* (Chl dg) 680 nm
 112. *Chlorophyll dh* (Chl dh) 680 nm
 113. *Chlorophyll di* (Chl di) 680 nm
 114. *Chlorophyll dj* (Chl dj) 680 nm
 115. *Chlorophyll dk* (Chl dk) 680 nm
 116. *Chlorophyll dl* (Chl dl) 680 nm
 117. *Chlorophyll dm* (Chl dm) 680 nm
 118. *Chlorophyll dn* (Chl dn) 680 nm
 119. *Chlorophyll do* (Chl do) 680 nm
 120. *Chlorophyll dp* (Chl dp) 680 nm
 121. *Chlorophyll dq* (Chl dq) 680 nm
 122. *Chlorophyll dr* (Chl dr) 680 nm
 123. *Chlorophyll ds* (Chl ds) 680 nm
 124. *Chlorophyll dt* (Chl dt) 680 nm
 125. *Chlorophyll du* (Chl du) 680 nm
 126. *Chlorophyll dv* (Chl dv) 680 nm
 127. *Chlorophyll dw* (Chl dw) 680 nm
 128. *Chlorophyll dx* (Chl dx) 680 nm
 129. *Chlorophyll dy* (Chl dy) 680 nm
 130. *Chlorophyll dz* (Chl dz) 680 nm
 131. *Chlorophyll ea* (Chl ea) 680 nm
 132. *Chlorophyll eb* (Chl eb) 680 nm
 133. *Chlorophyll ec* (Chl ec) 680 nm
 134. *Chlorophyll ed* (Chl ed) 680 nm
 135. *Chlorophyll ee* (Chl ee) 680 nm
 136. *Chlorophyll ef* (Chl ef) 680 nm
 137. *Chlorophyll eg* (Chl eg) 680 nm
 138. *Chlorophyll eh* (Chl eh) 680 nm
 139. *Chlorophyll ei* (Chl ei) 680 nm
 140. *Chlorophyll ej* (Chl ej) 680 nm

Der Algorithmus lässt sich also in drei Teile unterteilen, die einzeln optimierbar sind. Die Optimierungsergebnisse werden an die jeweils nächste Stufe weitergereicht.

- Der Verfahrensschritt 1 erfolgt in einem eigenem Durchlauf durch das Bild, da für den folgenden Verfahrensschritt das Kantenbild in der Umgebung schon bekannt sein muß. Die Approximation der realen Geraden und das anschließende Mischen der Farben kann hingegen in einem Durchlauf geschehen, da nur aufgrund der Lage der Geraden in diesem Punkt die Mischung erfolgt.

Für eine Hardware-Realisierung ist es dagegen so, daß bereits beim ersten Durchlauf das komplette Bild einmal aus dem Speicher ausgelesen werden muß, woraufhin dann das Kantenbild (mindestens 1 Bit pro Pixel) im Speicher abgelegt wird. Beim zweiten Durchlauf werden dann die Kanten in voller Länge im Kantenbild verfolgt, was bei einer beliebigen Richtung der Kante einen (fast) zufälligen Zugriff auf den Speicher zur Folge hat. Es ergeben sich also Probleme bei der Bearbeitungszeit, aufgrund der Notwendigkeit zweier getrennter Durchläufe, und des schwierigen Zugriffs auf den Speicher beim zweiten Durchlauf. Unschön ist natürlich auch, daß extra ein Speicherbereich zur Verfügung stehen muß, in dem das Kantenbild abgelegt werden kann, da es bei einer Größe von minimal 400 kByte (1 Bit bei einer Bildschirmauflösung von 2048 x 1536) schlecht im Chip gehalten werden kann.

Das erfindungsgemäße Verfahren bietet die Möglichkeit der Lösung dieser Probleme, indem die Kanten nur lokal verfolgt werden. Vorteilhaft ist es, die Kante nur innerhalb eines lokalen Fensters von $n \times n$ Pixeln im Bild zu verfolgen. Nur in diesem Fenster muß die Kanteninformation bekannt sein, die Speicherung des kompletten Kantenbildes entfällt also.

Bisherige Verfahren arbeiteten immer während der Generierung der Szenen und erforderten damit einen hohen Speicheraufwand und/oder eine Menge Rechenzeit. Durch die Verlagerung konnte dieser Nachteil größtenteils beseitigt werden, wobei

Da mit dem hier beschriebenen Antialiasing-Verfahren nicht jene Fälle behandelt werden können, für die im gepointsampelten Bild nicht genügend Informationen vorhanden sind, ist es weiterhin günstig, dieses Verfahren mit einem Anderen zu kombinieren. Ideal wäre ein Verfahren, welches ausschließlich Polygone (z.B. durch Supersampling oder Areasampling) behandelt, deren Breite bzw. Höhe unter zwei Pixel liegt, damit der zusätzliche Aufwand sich in Grenzen hält und die Echt-

Andere vorteilhafte Weiterbildungen der Erfindung sind in den Unteransprüchen gekennzeichnet bzw. werden nachstehend zusammen mit der Beschreibung der bevorzugten Ausführung der Erfindung anhand der Figuren näher dargestellt. Es zeigen:

- Zur Beschreibung der Verfahrenseinzelheiten soll zunächst der Verfahrensablauf des erfindungsgemäßen "Post-Anti-Aliasing" anhand von Pixeldarstellungen der zu betrachtenden Kantenbereiche näher beschrieben werden.

Signaltechnisch lässt sich ein Kantenbild durch die Faltung des Bildes mit einer Maske, deren Summe Null ist, gewinnen. Die Summe der Elemente in der Maske muß Null ergeben, da sich nur somit der Gleichanteil im Bild entfernen lässt (d.h. einfarbige Flächen werden im Kantenbild einen Wert von Null erhalten). Um diese Bedingung gewährleisten zu können, werden sowohl positive als auch negative Elemente in der Maske enthalten sein, so daß das ein durch Faltung erzeugtes (beispielsweise) Grauwertbild positive und negative Werte beinhalten wird. Der Betrag des Grauwertes ist dabei ein Maß für die Stärke einer Kante im betrachteten Pixel (ein Wert von Null bedeutet also, daß im Pixel keine potentielle Kante zu finden ist).

Ausgangspunkt für die Bestimmung der Lage der realen Geraden bei dem erfindungsgemäßen Verfahren ist ein binäres Kantenbild, wobei die Kante immer nur in Abhängigkeit vom verwendeten Kantenerkennungsoperator korrekt gefunden werden kann. Jeder Kantenoperator hat seine Eigenheiten (ein/zwei Pixel breite Kanten, Aussehen des Bereiches zwischen zwei Sprüngen), die im Kantenbild richtig interpretiert werden müssen, um sinnvolle Ergebnisse zu erhalten.

Durch das folgend beschriebene Verfahren entsteht zunächst ein Kantenbild mit 4 Bit Information pro Pixel, das letztlich vorzustellende Verfahren arbeitet aber auf

Ausgehend von dem aktuell zu betrachtenden Pixel wird nun aus den noch vorhandenen zwei Bit die richtige Information ausgeblendet. Ist das Zentralpixel als positiv markiert, so werden auch in der Umgebung nur positive Kantenpixel berücksichtigt, um wirklich nur die positive Kante zu verfolgen; entsprechend werden, falls das Zentralpixel als negativ markiert ist, nur negative Kantenpixel betrachtet. Somit entfällt beim Beispiel eines senkrecht stehenden Fahnenmastes die störende negative bzw. positive Kante und beide Fälle können korrekt behandelt werden, wie es in Figur 2 dargestellt ist. Hier wird die Behandlung zweier nebeneinanderliegender Kanten dargestellt, die nach Rückführung auf den horizontalen Fall durch Drehung als x-dominante Binärbilder weiterverarbeitet werden. Ist das Zentralpixel jedoch als positiv und negativ markiert, so liegt der Spezialfall vor, bei dem benachbarte Farben die gleiche Farbvektoralänge besitzen,

und es erfolgt eine ODER-Verknüpfung der beiden Informationen pro Pixel.

Von nun an wird also mit einem binären Kantenbildausschnitt weitergearbeitet, bei dem nur noch x-dominante Kanten markiert sind, d.h., der Winkel der zu erkennen- den Geraden liegt zwischen -45 und $+45^\circ$.

Um aus dem vorliegenden Kantenbild reale Geraden extrahieren zu können, muß erst einmal ermittelt werden, wie die dargestellten Pixel generiert wurden.

Die Geometriebeschreibung einer in einem Bild dargestellten Szene erfolgt fast ausschließlich auf Polygonbasis, da Polygone am einfachsten handhabbar sind. Weitere Darstellungsmöglichkeiten, wie B-Splines (Angabe nur von Stützstellen der Oberfläche), Voxelm Modelle (Beschreibung des kompletten Volumens des Objektes), sowie CSG-Bäume (Aufbau des Objektes aus geometrischen Primitiva), finden kaum Anwendung, da aus ihnen abgeleitete Darstellungsverfahren für die Rasterisierung zu komplex sind, um sie in Hardware realisieren zu können.

Bei den in Hardware realisierten Polygonrenderern handelt es sich meist um Dreiecksrenderer, da nur bei Dreiecken gewährleistet ist, daß nach einer Transformation (Rotation, Skalierung, Translation) das Polygon eben bleibt, was bei allgemeinen Polygonen aufgrund der beschränkten Rechengenauigkeit der Rechner nicht immer der Fall ist.

Die Dreiecke, beschrieben durch die drei Eckpunkte und Parametern (z-Wert, Texturkoordinaten, Normalen, usw.) an diesen Punkten, werden in einem sogenannten "Scanliner" rasterisiert, und anschließend wird jedem Pixel aufgrund der interpolierten Parameter ein Farbwert zugewiesen, der dann im Framebuffer abgelegt wird, falls das Pixel nicht durch ein Pixel eines anderen Dreiecks verdeckt ist. Unser Interesse gilt nun genau diesem Scanliner, der entscheidet, welche Pixel gesetzt werden, und welche nicht.

Die Arbeitsweise eines Scanliners läßt dabei sich wie folgend beschreiben: Ausgehend vom unteren Eckpunkt des Dreiecks wird für jede Zeile der reale Anfangs- und Endpunkt bestimmt, alle Pixel die zwischen diesen beiden Punkten liegen, gehören demnach zum Dreieck und werden dargestellt. Die Entscheidung, welches Pixel das erste bzw. letzte darzustellende ist, wird aufgrund der Lage des Pixelmittelpunktes zur realen Kante getroffen. Liegt der Pixelmittelpunkt innerhalb des Anfangs- und Endpunktes dieser Zeile, so wird das Pixel gesetzt. Durch dieses

Ergibt sich im Bild eine Stufe, so ist bekannt, daß die reale Kante zwischen den beiden Pixelmittelpunkten der verschiedenfarbigen Pixel verlaufen muß. Dies ist in Figur 3 dargestellt. Werden mehrere Stufen bei diesem Vorgehen berücksichtigt, so wird immer genauer die reale Kante angenähert. Die Verfolgung von Stufen gestaltet sich in realen Bildern schwieriger als hier zunächst angenommen, da im Bildausschnitt meist nicht nur eine Kante enthalten ist, sondern auch sich kreuzende Kanten, Kanten, die ihre Steigung ändern, sowie Texturkanten und Rauschen.

Über das aktuelle Fenster des Kantenbildes wird der Einfachheit halber ein lokales Koordinatensystem gelegt, durch das alle Pixel von nun an beschrieben werden können. Das Zentralpixel erhält die Koordinate (0/0.5); 0.5 in der y-Koordinate deshalb, da der hier angenommene Differenzoperator immer das Pixel oberhalb des Farbsprunges markiert, bei $y=0.0$ liegt demnach genau der Farbsprung. Ein derartiges Fenster ist in Figur 4 dargestellt.

Ist das Zentralpixel also als Kante markiert, so wird im aktuellen Fenster die Stufe weiter nach rechts bzw. links verfolgt, um die nächstgelegenen Sprünge zu finden. Es ergeben sich somit zwei Werte x_A und x_E , die die Endpunkte der Stufe bezeichnen. Am Ende der Stufe kann es nun prinzipiell vier verschiedene Konstellationen geben.

1. Schräg oberhalb liegt ein weiteres Kantenpixel, d.h., es erfolgt ein Sprung nach oben (Dieser Fall wird weiterhin als UP bezeichnet.)
2. Schräg unterhalb liegt ein weiteres Kantenpixel, d.h., es erfolgt ein Sprung nach unten (Fall: DOWN)
3. In der nächsten Spalte existiert kein weiteres Kantenpixel; die Kante hört also hier vollkommen auf (wir befinden uns wahrscheinlich an einer Ecke des Objektes) (Fall: NO)

- 100° 110° 120° 130° 140° 150° 160° 170° 180° 190° 200° 210° 220° 230° 240° 250° 260° 270° 280° 290° 300° 310° 320° 330° 340° 350° 360°

Falls an beiden Enden der Stufe entgegengesetzte Fälle auftreten (links UP / rechts DOWN oder links DOWN / rechts UP), kann nun bereits eine Gerade festgelegt werden. Die Endpunkte befinden sich bei $x_{\text{Anf}} := x_A - 0.5$ und $x_{\text{End}} := x_E + 0.5$ und die y-Koordinate ergibt sich bei einem Sprung nach oben zu 0.5, bei einem Sprung nach unten zu -0.5.

Am Beispiel einer schräg nach oben verlaufenden Kante soll verdeutlicht werden, warum die Gerade durch die beiden Punkte

$$(x_{\text{Anf}}, y_{\text{Anf}}) = (x_A - 0.5, -0.5) \text{ und} \\ (x_{\text{End}}, y_{\text{End}}) = (x_E + 0.5, 0.5)$$

gelegt wurde. Zu beachten ist dabei die Unterscheidung von x_A (letztes Pixel der Stufe) bzw. x_E und x_{Anf} (Punkt auf der angenommenen Geraden) bzw. x_{End} . Aufgrund des Musters im Kantenbild ist genau bekannt, wo sich der Sprung zwischen den beiden beteiligten Farben befindet. Dieser Zusammenhang ist aus Figur 6 ersichtlich, welche die Zuordnung der möglichen Verläufe der realen Geraden zu einer Kante wiedergibt (Links oben: Muster im Kantenbild, rechts oben: Farbzuordnung der Pixel mit angenommener Geraden), links unten: minimale und maximale Steigung, rechts unten: parallele Geraden. Irgendwo zwischen den horizontal andersfarbigen Pixeln muß also die reale Kante verlaufen.

Während in Figur 6 die möglichen Extremfälle der möglichen Geraden dargestellt sind, bildet die gewählte Gerade nun genau einen Mittelweg zwischen allen Extremen, so daß der Fehler zur realen Geraden, unabhängig davon, um welche es sich handelt, minimiert wird.

Bei der Behandlung des Falles NO, in dem kein weiteres Kantenpixel in der nächsten Spalte vorhanden ist, wird dem Endpunkt die y-Koordinate 0.0 zugewiesen, da aus dem Kantenbild nicht ersichtlich ist, warum kein weiteres Pixel existiert. Der Grund kann entweder sein, daß eine Ecke eines Objektes vorliegt (in diesem Fall erscheint die Ecke etwas abgerundet), oder aber der Schwellwert bei der Kantenbildgenerierung nicht klein genug angesetzt war (so daß die Kante beliebig weiterläuft, jedoch nicht mehr als solche erkannt wurde).

Im vierten Fall (HOR) muß eine Sonderbehandlung vorgesehen werden, da dieser nur aufgrund der Beschränkung des Algorithmus auf ein Fenster auftritt.

Der einfachste Ansatz ist, diesen Fall genau wie den vorherigen zu behandeln, dem entsprechenden Endpunkt also den y-Wert 0.0 zuzuweisen. Dies würde jedoch zu unrichtigen Ergebnissen führen, wie am Beispiel von Figur 7 ersichtlich ist. Für jedes Pixel wurde die Gerade eingezeichnet, die sich durch dieses Vorgehen ergibt. Bei der Verschiebung des Fensters wird die erzeugte Gerade immer flacher, bis sie auf einmal vollkommen horizontal wird, sobald der Sprung aus dem Fenster läuft.

Am Ende einer Stufe kann es vorkommen, daß sowohl ein Sprung nach oben als auch ein Sprung nach unten angezeigt wird, was durch das Zusammenlaufen zweier Kanten entsteht. Dieser Fall ist in Figur 10 wiedergegeben. Dabei ist im linken Teil der Figur das Kantenbild und im rechten Teil die Ausgangssituation dargestellt, welche zu dem links dargestellten Bild führte. Als beste Lösung für diesen Fall stellte sich die Betrachtung des entgegengesetzten Status heraus. Ist der

Der Einfachheit halber wird für die Betrachtung ein anderes Koordinatensystem zugrunde gelegt, bei dem der Nullpunkt im Pixel links von der Stufe liegt. Dies ist

aus Figur 12 ersichtlich.

Die reale Gerade verläuft also sicher zwischen den Koordinaten (0,0) und (1,0) bzw. $(x_E, 1)$ und $(x_E + 1, 1)$, da sich sonst eine andere Rasterisierung ergeben hätte. Die angenommene Gerade g verläuft durch die Punkte (0.5,0) und $(x_E + 0.5, 1)$, woraus sich die Geradengleichung ergibt zu

$$g: y = \frac{1}{x_E} * (x - \frac{1}{2})$$

Wesentlich ist die maximale Differenz dieser Geraden zu allen anderen Möglichen. Die Differenz wird nur an den ganzzahligen x-Koordinaten benötigt, da nur dort Samplepoints liegen. Es müssen also die Differenzen zu allen möglichen Extremfällen der realen Geraden betrachtet werden.

Es ergeben sich drei Fälle (siehe Figur 6)

1. Differenz zu allen parallelen Geraden

Falls die reale Gerade parallel zur Angenommen verläuft, so ist die Differenz für jedes Pixel der Stufe die selbe. Aus der Geradengleichung für die am weitesten unten möglichen Geraden (Punkte (1,0) und $(x_E + 1, 1)$)

$$g_u: y = \frac{1}{x_E} * (x - 1)$$

ergibt sich

$$\Delta y_1 = |g - g_u| = \frac{1}{2 * x_E}$$

als Differenz bei jedem x-Wert.

Aus der Geradengleichung für die am weitesten oben liegenden Geraden (Punkte (0,0) und $(x_E, 1)$)

$$g_0: y = \frac{1}{x_E} * x$$

ergibt sich

$$\Delta y_2 = |g_0 - g| = \frac{1}{2 * x_E}$$

als Differenz bei jedem x-Wert.

2. Differenz zur Geraden mit maximaler Steigung

Die Gerade mit der maximalen Steigung verläuft durch die Punkte (1,0) und $(x_E, 1)$ und wird beschrieben durch:

$$g_M: y = \frac{1}{x_E - 1} * (x - 1)$$

$$\Delta y_3(x) = |g_M - g| = \left| \frac{1}{(x_E - 1) * x_E} * \left(x - \frac{1}{2} * x_E - \frac{1}{2} \right) \right|$$

Die maximale Differenz wird sich am weitesten außen ergeben, da dort die Geraden immer weiter auseinanderlaufen, es werden also die x-Werte $1(\Delta y_3(1) = 1/2 x_E)$ und $x_E(\Delta y_3(x_E) = 1/2 x_E)$ in die Gleichung 10 eingesetzt, wodurch sich die maximale Differenz

$$\Delta y_3 = \frac{1}{2 * x_E}$$

ergibt.

3. Differenz zur Geraden minimaler Steigung

Die Gerade mit der minimalen Steigung verläuft durch die Punkte (0,0) und $(x_E + 1, 1)$ und die Geradengleichung lautet demzufolge:

$$g_m: y = \frac{1}{x_E + 1} * x$$

$$\Delta y_4(x) = |g_m - g| = \left| \frac{1}{(x_E + 1) * x_E} * \left(-x * \frac{1}{2} * x_E + \frac{1}{2} \right) \right|$$

Nach den gleichen Überlegungen wie bei Punkt 2 ergeben sich die maximalen Differenzen bei 1 () und x_E (), wodurch die maximale Differenz

$$1 \quad (\Delta y_4(1) = \frac{(x_E - 1)}{2 * (x_E + 1) * x_E} \quad \text{und} \quad x_E \quad (\Delta y_4(x_E) = \frac{(x_E - 1)}{2 * (x_E + 1) * x_E})$$

entsteht.

Wie aus den obigen Gleichungen ersichtlich ist, beträgt die maximale Differenz zu den Geraden $\Delta y = x/2x_E$ Figur 13 gibt die maximale Abweichung in Abhängigkeit von der Stufenlänge x_E . Für lange Stufen entspricht die angenommene Gerade praktisch der wirklichen, wohingegen für kurze Stufen ($x_E < 4$) kaum annehmbare Differenzen entstehen, wenn man vom Grenzfall einer Kante zwischen weiß und schwarz ausgeht.

Eine Verbesserung ergibt sich, wenn statt nur über eine Stufe die Kante über mehrere Stufen verfolgt wird, da dadurch die reale Gerade besser approximiert werden kann. Bei einer fest vorgegebenen Fenstergröße kann bei kurzen Stufen eine Interpolation über viele Stufen erfolgen, bei langen Stufen meist nur eine Stufe interpoliert werden, was der Form der Differenz-Funktion entgegenkommt. Bei sehr langen Stufen, die nicht mehr komplett in das aktuelle Fenster hineinpassen, wird nicht mehr versucht, der realen Geraden zu folgen, da dazu lokal die Information fehlt, so daß ab einer Stufenlänge, die über die Fensterbreite hinausgeht, jede Stufe die gleiche Interpolation erfährt (Figur 8). Das Fenster sollte also möglichst groß gewählt werden, um diesen Effekt zu vermeiden; bei einem Fenster der Größe des eigentlichen Bildes tritt dieser Fehler gar nicht mehr auf. Zu große Fenstergrößen widersprechen aber der eigentlichen Idee des Algorithmus, so daß ein Kompromiß zwischen den beiden Extremen gefunden werden muß.

Bei der Verfolgung nur einer Stufe tritt wie gezeigt das Problem auf, daß kurze Stufen zu einer sehr ungenau approximierten Geraden führen. Dies wird besonders augenfällig, wenn die Rasterisierung einer Kante nahe 45° betrachtet wird, bei der abwechselnd "ein-pixel-lange" und "zwei-pixel-lange" Stufen auftreten (Figur 14). Bei den "ein-pixel-langen" Stufen wird eine Gerade der Steigung 45° angelegt, bei den "zwei-pixel-langen" Sprüngen jedoch eine Gerade mit der Steigung

$$\arctan \frac{1 \text{ (Stufen in y-Richtung)}}{2 \text{ (Stufen in x-Richtung)}} \approx 26,6 \text{ Grad}$$

Durch die Verfolgung mehrerer Stufen kann die Vielfalt der möglichen Geraden (Steigung zwischen $26,6$ und 45°) auf einen kleinen Bereich eingeschränkt werden,

Der einfachste Ansatz wäre jetzt, die jeweiligen Endpunkte der letzten Stufen als Punkte auf der approximierten Geraden festzulegen, jedoch liefert das in einigen

Fällen vollkommen falsche Ergebnisse. Zum Beispiel wird der Fall zweier Kanten unterschiedlicher Steigungen nicht korrekt behandelt. Durch das hier skizzierte Verfahren würde die Ecke zwischen den beiden Kanten nicht wahrgenommen, und somit weginterpoliert, wobei nicht einmal eine kontinuierliche Änderung entsteht, sondern immer wieder Sprünge (Figur 16). Noch ungünstiger sieht es bei der Rasterisierung eines Kreises aus, der prinzipiell aus mehreren Kanten zusammengesetzt werden kann, deren Steigungen sich allmählich ändern. Hierbei würde sogar über mehrere Ecken hinweg interpoliert werden.

Bei der Rasterisierung durch die Scankonvertierung entstehen an einer Kante charakteristische Sprungmuster, die eingehalten werden müssen. Ist für einen Sprung eine der Regeln nicht mehr gültig, so kann man davon ausgehen, daß dieser Sprung zu einer anderen Kante gehört, somit kann die entsprechende Stufe eliminiert werden.

Regeln bei der Rasterisierung einer Kante:

1. Es können maximal zwei verschiedene Stufenlängen vorkommen, deren Stufenlängen sich außerdem um maximal 1 unterscheiden dürfen.
2. Nur eine der beiden Stufenlängen darf mehrmals hintereinander auftreten.
3. Durch das Aneinanderreihen der Anzahlen der Stufen, die die gleiche Länge haben, erhält man eine Zahlensequenz, bei der abwechselnd immer eine Eins und dann eine beliebige Zahl (> 0) steht. Die Einsen (nur die an jeder zweiten Position) werden aus dieser Sequenz gestrichen. Bei der erhaltenen Sequenz dürfen wieder nur zwei verschiedene Zahlen vorkommen, die sich um eins unterscheiden.
4. Bei der unter 3. erhaltenen Sequenz darf nur eine der beiden möglichen Zahlen mehrmals hintereinander auftreten.
5. Durch wiederholtes Anwenden der Regeln 3. und 4. auf die Zahlensequenz, läßt sich ein immer globalerer Blick auf die Kante gewinnen.

Je kleiner das betrachtete Fenster gewählt wird, desto weniger der oben genannten Regeln müssen beachtet werden, da in einem kleinen Fenster nur sehr wenige Stufen passen, und somit nicht genügend Information für alle Regeln zur Verfügung

steht. Bei der hier als Beispiel beschriebenen Hardware-Implementierung wurde eine Fenstergröße von 17×17 gewählt, bei der nur die ersten beiden Regeln implementiert wurden, da die dritte Regel nur in sehr wenigen Fällen Anwendung finden würde, und die Unterschiede so marginal sind, daß der Mehraufwand nicht gerechtfertigt ist. Bei der nachfolgend beschriebenen Software-Implementierung, die für beliebige Fenstergrößen ausgelegt ist, wurden die Regeln bis einschließlich Nr. 4 berücksichtigt.

Ausgehend von der zentralen Stufe werden sukzessive jeweils immer eine Stufe nach rechts bzw. links zu den betrachteten Stufen hinzugefügt. Ist für eine der zugefügten Stufen eine Regel nicht mehr erfüllt, so wird die Stufe wieder entfernt, und es wird nur noch in die andere Richtung weitergesucht. Können auf beiden Seiten keine weiteren Stufen hinzugenommen werden (ohne die Regeln zu verletzen), so ist dieser Vorgang abgeschlossen, und es können aus den Stufen die Endpunkte der anzulegenden Geraden bestimmt werden.

Falls die letzten Stufen an beiden Enden komplett benutzt werden, ergibt sich das Problem, daß es bei der Verschiebung des Fensters um ein Pixel vorkommen kann, daß am linken Rand eine Stufe nicht mehr ins Fenster paßt, und gleichzeitig eine neue Stufe am rechten Rand dazugenommen wird, was in einer relativ starken Änderung der angenommenen Geraden resultiert.

In Figur 18 ist wiedergegeben, wie die Verschiebung des Fensters um ein Pixel nach rechts zu einer neuen und einer wegfallenden Stufe führt.

Dieser Effekt kann vermieden werden, wenn die letzten Stufen nur halb benutzt werden, das heißt es wird die halbe Länge zur x-Koordinate und 0.5 zur y-Koordinate hinzugerechnet. Durch dieses Vorgehen wird die abrupte Hinzunahme bzw. das abrupte Weglassen einer Stufe verdeckt, und die Geraden benachbarter Pixel passen sich besser einander an (Figur 19).

Das Verfahren arbeitet wie vorgestellt auf einem gepointsamplen Bild, die endgültige Farbgebung eines Pixels ergibt sich also aus seiner gepointsamplen Farbe unter Zumischung der Farben benachbarter Pixel. Die Zumischung der Farben wird jedoch nicht in jedem Fall durchgeführt. Handelt es sich bei betrachteten Pixel um kein Kantenpixel, das heißt es wurden keine extremen Farbsprünge zu den benachbarten Farben wahrgenommen, so wird der ursprüngliche Farbwert übernommen, was bei ca. 90% der Pixel eines Bildes auftritt. In den meisten

Die Bestimmung der einzelnen Mischfaktoren erfolgt auf der Grundlage der pro Pixel angelegten Gerade(n), die durch die wahrnehmbaren Farbsprünge im Bild bestimmt werden. Die Zumischungsrichtung ergibt sich aus der Lage des Pixels bezüglich des Farbsprunges. Wurde beispielsweise ein Farbsprung zum darüberliegenden Pixel erkannt, so wird daraus ein Mischfaktor bzgl. dieses Pixels bestimmt.

1. Ist das darüberliegende Pixel im Kantenbild als horizontal markiert, so ergab sich ein Farbsprung zum aktuellen Pixel und es wird durch dieses Pixel eine Gerade bestimmt, die den UP-Mischfaktor festlegt.
2. Ist das aktuelle Pixel im Kantenbild als horizontal markiert, so ergab sich ein Farbsprung zum darunterliegenden Pixel und es wird eine Gerade bestimmt, die den DOWN-Mischfaktor festlegt.
3. Ist das linke Pixel als vertikal im Kantenbild markiert, so ergab sich ein Farbsprung zum aktuellen Pixel und es wird eine Gerade unter Rückführung auf den horizontalen Fall bestimmt, die den LEFT-Mischfaktor festlegt.
4. Ist das aktuelle Pixel als vertikal markiert, so ergab sich ein Farbsprung zum rechten Nachbarn und es wird eine Gerade unter Rückführung auf den horizontalen Fall bestimmt, die den RIGHT-Mischfaktor festlegt.

Bei den Punkten 1 und 3 fällt auf, daß das aktuelle Pixel gar nicht als Kantenpixel markiert worden war. Das kam daher, daß der Differenzoperator immer nur Pixel markiert, die entweder links (für den vertikalen Operator) oder oberhalb (für den horizontalen Operator) des Farbsprunges liegen. In diesen beiden Fällen muß das aktuelle Fenster also um ein Pixel nach oben (im Fall 1) bzw. um ein Pixel nach links (im Fall 3) verschoben werden, damit das nachfolgend vorgestellte Verfahren angewendet werden kann, da dort davon ausgegangen wird, daß das Zentralpixel ein Kantenpixel enthält, von dem aus die Gerade verfolgt werden soll.

Liegt der Betrag des y-Wertes zwischen 0.5 und 1.0, so sollte aufgrund der Geraden das Pixel bereits den anderen Farbwert haben, nach dem Kantenbild hat es ihn jedoch nicht. Dies kann dadurch zustande kommen, daß die angenommene Gerade nicht exakt der wirklichen Kante entspricht. Es werden jedoch trotzdem Faktoren bis 1.0 zugelassen, da benachbarte Pixel ebenfalls nach der vermeintlich

Das zweite Verfahren liefert genauere Ergebnisse. Hierbei wird aufgrund der Lage der Geraden bezüglich des Pixelmittelpunktes der genaue Flächenanteil bestimmt, der sich unterhalb der Geraden befindet (Figur 20).

Die Flächenanteile lassen sich mittels eines Tabellen-Lookups bestimmen, bei dem lediglich die Steigung der Geraden und der Abstand der Geraden zum Pixelmittelpunkt eingeht. Das Verfahren läßt sich also recht gut in Hardware realisieren und stellt damit eine Erweiterungsoption des hier vorgestellten Algorithmus dar, mit dem eine größere Genauigkeit erzielt werden kann, jedoch mit einem größeren Hardwareaufwand, da zusätzliche Tabellen benötigt werden (Figur 21).

Als problematisch stellt sich die Bestimmung des Mischfaktors für das zentrale Pixel dar. Falls nur horizontale Mischfaktoren (also UP und DOWN) bzw. nur vertikale Mischfaktoren (LEFT und RIGHT) ungleich Null sind, ergibt sich der zentrale Mischfaktor zu

$$\text{mix}_{\text{center}} = 1 - \text{mix}_{\text{up}} - \text{mix}_{\text{down}} - \text{mix}_{\text{left}} - \text{mix}_{\text{right}}$$

Diese genauere Betrachtung macht aber nur Sinn, wenn die Mischfaktoren sich auch auf die wirklichen Flächenanteile beziehen. Da bei dem implementierten Verfahren dies nicht verwirklicht ist, wird hier erneut eine Näherung eingeführt, indem bei vertikalen und horizontalen Mischfaktoren alle Faktoren halbiert werden.

Durch diese Vorgehen ist wieder gewährleistet, daß der zentrale Mischfaktor im Bereich 0.0 bis 1.0 liegt. Die verwendete Näherung ist sehr grob, die Fehler werden im Bild jedoch nicht wahrgenommen, da die Ecke eines Objektes eine Diskontinuität darstellt, und dadurch der entstehende Farbfehler nicht auffällt.

Die Farbmischung eines Pixels erfolgt letztlich pro Farbkanal nach der Mischungsformel:





Die Implementierung des Post-Antialiasing-Verfahrens erfolgte in der Programmier-

Der Pseudoalgorithmus für das Verfahren stellt sich wie folgt dar (dick geschriebene Variablen sind die jeweiligen Ergebniswerte):

```
PostAntialias(Image pic, Image result) {

    generateEdgePic(pic, edges);

    for y in 0..(pic.height-1) {

        for x in 0..(pic.width-1) {

            if isEdgePixel(x,y) {

                cutActualWindow(edges, edgeWindow);

                computeMixFactorsHorizontal(x, y, edgeWindow,

                    mix_up, mix_down);

                transformVerticalToHorizontal(edgeWindow);

                computeMixFactorsHorizontal(x, y, edgeWindow,

                    mix left, mix right);
            }
        }
    }
}
```

```
mixColors(mix_up, mix_left, mix_right,

          mix_down, pic[x,y-1], pic[x-1,y],
          pic[x,y], pic[x + 1,y], pic[x,y + 1],

          mixedColor);

storeColor(mixedColor, result[x,y]);

} else {

    storeColor(pic[x,y], result[x,y]);

}

}

}
```

In `isEdgePixel()` wird getestet, ob das Pixel (x,y) als Kante, das Pixel $(x,y-1)$ als horizontale oder das Pixel $(x-1,y)$ als vertikale Kante im Kantenbild markiert ist; nur dann muß eine weitere Behandlung erfolgen, ansonsten wird mittels `storeColor()` der alte Farbwert einfach ins neue Bild übernommen.

In `generateEdgePic()` wird mittels des beschriebenen Verfahrens das Kantenbild generiert. In `cutActualWindow()` wird aus dem Kantenbild das Fenster ausgeschnitten, welches als Umgebung für das aktuelle Pixel dient. Dies ist nötig, da in der späteren Hardware auch nur eine begrenzte Umgebung zur Verfügung steht, da das Kantenbild "on-the-fly" generiert wird und nicht wie hier im voraus.

Die Funktion `computeMixFactorsHorizontal()` wird sowohl für die horizontalen Mischfaktoren (`mix_up` und `mix_down`) als auch für die vertikalen (`mix_left` und `mix_right`) aufgerufen. Dies ist möglich, da das aktuelle Kantenfenster mittels `transformVerticalToHorizontal()` so umgebaut wird, daß die ehemals vertikalen Kanten nun horizontal verfolgt werden können.


```
} else {  
    mix_up = 0;  
  
}  
  
}
```

Mittels `isEdge()` wird getestet, ob an der entsprechenden Stelle im Kantenfenster ein Kantenpixel vorliegt, nur dann kann nämlich eine Kante verfolgt werden, ansonsten wird der entsprechende Mischfaktor gleich auf Null gesetzt. In `maskRightEdgeType()` wird aus dem Kantenfenster, welches bisher noch 4 Bit pro Pixel enthielt, das Bit ausgeblendet, welches durch das Zentralpixel des Fensters vorgegeben wird, so daß ab sofort mit einem Binärbild weitergearbeitet wird. Mittels `translateWindow()` werden alle Pixel des Kantenfensters um ein Pixel nach unten verschoben, so daß im Zentrum sich das Kantenpixel befindet, von dem aus die Kante verfolgt werden soll, um den Mischfaktor für das obere Pixel zu erhalten.

In `getPositionInEdge()` können nun beide Fälle genau gleich behandelt werden. In dieser Funktion erfolgt die wirkliche Verfolgung der Kante im Kantenbild, was zum Anfangspunkt ($x_{\text{Anf}}, y_{\text{Anf}}$) und Endpunkt ($x_{\text{End}}, y_{\text{End}}$) und dem jeweiligen Status (UP / DOWN / NO / HOR) an den Enden führt.

Aufgrund dieser Information wird dann in `mixFactor()` letztlich der Mischfaktor bestimmt, der sich aus der jeweiligen Kante ergibt.

In `mixColors()` wird zunächst der noch fehlende Mischfaktor für das zentrale Pixel bestimmt, wobei wiederum verschiedene Varianten vorgesehen sind, und dann erfolgt anhand der Mischungsformel die wirkliche Mischung der fünf in Frage kommenden Farben.

Die nachfolgend dargestellte Hardware-Implementierung ist ein Ausführungsbeispiel, welches primär dazu dient die Ausführung des Verfahrens zu veranschaulichen. Es wurde versucht, die Lösung so allgemein wie möglich zu gestalten. So wurden beispielsweise die Adreßbreiten für die Zähler variabel gehalten, sowie auch die Bitbreiten der Mischfaktoren, genau wie die Tabellenbreite und -länge der verwendeten Dividierer. Die Allgemeinheit konnte jedoch nicht überall beibehalten werden, da sonst einige Module zu komplex geworden wären. Als Beispiel ist das Modul `findSeq()` zu erwähnen, bei dem ausgehend von einem ($n \times n$)-Fenster etwa

$n^2/8$ Pixel zu berücksichtigen sind, um die Verfolgung der Stufen zu gewährleisten. Weiterhin mußten die Positionen der Registerstufen auf einen speziellen Fall festgelegt werden, da bei einem größeren Fenster mehr Pipeline-Stufen nötig sind, weil die Komplexität ansteigt. Der Übersichtlichkeit halber wurden in den Blockschaltbildern der parametrisierbaren Module konkrete Bitbreiten vorgegeben.

Für die Hardware-Implementierung wurden folgende Parameter und Grenzwerte gewählt:

- Fenstergröße: 17 x 17

Ist die Fenstergröße zu klein gewählt (etwa 9 x 9), so läßt die Bildqualität bei flachen Kanten zu wünschen übrig, da die Übergangsbereiche sehr kurz werden (nur 8 Pixel / vergleiche dazu Figur 8). Ist das Fenster jedoch zu groß gewählt, so steigt die Komplexität einiger Module, und damit auch die Gatteranzahl, sehr stark an. Für die Implementation wurde also eine mittlere Fenstergröße gewählt, um einen Kompromiß zwischen diesen beiden Extremen zu finden.

- maximale Bildschirmbreite: 2048 Pixel/Zeile

Die maximale Bildschirmbreite wurde mit 2048 Pixel pro Zeile angenommen. Dies stellt sicherlich eine obere Grenze für Bildschirmauflösungen der nächsten Jahre dar. Die maximale Bildschirmhöhe ist ebenfalls mit 2048 Zeilen festgelegt. Da das Verhältnis von Breite zu Höhe im Regelfall fest vorgegeben ist (1,25 oder 1,33), so wird dieser Maximalwert sicherlich nie erreicht werden. Die Festlegung der maximalen Bildschirmbreite führte zum Festlegen der Adreßbreiten der Zähler auf 11 Bit und der maximalen Länge der verwendeten Pipes auf 2048 Elemente.

- Bitbreite der Mischfaktoren: 7 Bit

Dieser Wert ergab sich aus der Fehlerbetrachtung der maximalen Abweichungen der Mischfaktoren vom tatsächlichen Wert. (Die maximale Abweichung beträgt maximal $1/256$ des Original-Mischfaktors. Aufgrund der Form der Mischungsformel ergibt sich somit eine maximale Abweichung um 5 Farbwerte.)

Aus der Festlegung der Mischfaktorbreite ergab sich auch die nötige Genauigkeit der Dividierer, die mit einer Bitbreite von 9 Bit angenommen wurden.

Das Antialiasing-System() gemäß Figur 24 sowie Tabelle 4 ist das Hauptmodul, von dem aus alle weiteren Module importiert werden und als Teilkomponenten weiter unten beschrieben sind.

1. Generierung der Kanteninformation

- Da die Mischung auf dem Kantenfenster arbeitet, muß der Kantengenerierungsprozeß immer einen Vorsprung von 8 Zeilen und 8 Punkten haben, so daß bei einem 17 x 17 Fenster schon alle Kanteninformationen zur Verfügung stehen, wenn sie benötigt werden.

Die Kanteninformation wird im Modul EdgeValueGenerator() (Tabelle 6, Figur 27) aus den neu eingelesenen Farbwerten erzeugt. Im Modul WindowSwitch() (Figur 31) wird der jeweils aktuelle Ausschnitt aus dem Kantenbild gehalten. Im Modul CenterPipe() werden die temporär nicht mehr benötigten Kanteninformationen zwischengespeichert. Pro Pixel werden 4 Bit Kanteninformation gehalten, wodurch sich bei 16 parallel abgelegten Pixeln eine Bitbreite von 64 Bit ergibt. Das wichtige bei diesem Modul ist, daß die Kanteninformation "on-chip" gespeichert wird, denn in jedem Takt werden 64 Bit abgelegt und auch parallel 64 Bit wieder eingelesen. Ein externer Speicher wäre also ständig damit beschäftigt, Werte abzulegen und wieder zu holen, und könnte somit für nichts Anderes verwendet werden. Die Speicherung ist praktisch auch möglich, da bei einer maximalen Bildschirmbreite von 2048 Pixeln 128 kB abgelegt werden müssen. (Derzeit sind on-chip RAMs von 1 MBit ohne größere Probleme realisierbar.) Dieser Speicher ist in beiden Varianten (Triple-Buffer / Display-Prozeß) nötig.

Für jedes einzelne Pixel müssen bis zu vier Geraden verfolgt werden, um die vier Mischfaktoren aus den einzelnen Richtungen zu bestimmen. Statistisch gesehen, wird in etwa 61% der Fälle nur eine Gerade benötigt. Jedoch kommen auch die anderen Fälle (zwei Geraden 27%, drei Geraden 8% und sogar vier Geraden 4%) vor. Da in jedem Takt ein neues Pixel bearbeitet wird, muß also der schlimmste Fall von vier Geraden angenommen werden und eine parallele Bearbeitung aller vier Fälle erfolgen. Dies geschieht in den Modulen `ComputeEdge()`, die als Ergebnis jeweils einen Mischfaktor für die entsprechende Gerade liefern. Für die vertikale Verfolgung der Geraden kann dabei das gleiche Modul wie für die horizontale Gerade verwendet werden, da, wie schon häufiger erwähnt, das vertikale Kantenfenster vorher zu einem horizontalen transformiert wird (geschieht in `WindowSwitch()`).

Bei genauer Betrachtung der zu verfolgenden Geraden fällt auf, daß diejenige Gerade, die für das aktuelle Pixel zum rechten Mischfaktor führt, mit der Geraden identisch ist, die beim nächsten Pixel zum linken Mischfaktor führt. Aufgrund dessen wird bei der Behandlung der rechten Geraden die Position bezüglich des aktuellen Pixels für das nächste Pixel weiterverwendet, so daß nicht vier, sondern nur drei Geraden verfolgt werden müssen; somit reduziert sich der Hardware-Aufwand. Die gleiche Optimierung ist für die obere bzw. untere Gerade nicht so einfach möglich, da zwar die gleiche Gerade verfolgt wird, die Information aus dieser Geraden aber über eine Zeile aufbewahrt werden müßte, was einen zusätzlichen Speicheraufwand von 32 kB bedeuten würde.

In `ColorPipe()` werden die Farbwerte aufgehoben, bis sie für die Mischung der neuen Farben wieder benötigt werden. In `CenterPipe()` bzw. `UpPipe()` werden die Farbwerte jeweils um eine zusätzliche Zeile verzögert. Der Inhalt der Pipes stellt sich wie in Figur 25 dar.

Der `edgecounter()` (Figur 26, Tabelle 5) gibt an, für welches Pixel gerade die Kanteninformation generiert wird. Dementsprechend werden bei einigen Pixeln die Nachbarn über die beiden Signale `rightValid` und `downValid` ausgeblendet, die im Bild keine wirklichen Nachbarn sind. `rightValid` wird jeweils beim letzten Pixel einer Zeile zurückgenommen, da das nächste Pixel schon zur nächsten Zeile gehört und somit nichts mit dem aktuellen Pixel gemeinsam hat. `downValid` wird in der letzten Zeile zurückgenommen, da die Zeile darunter keine gültigen Daten mehr enthält. `Ready` gibt an, wann alle Pixel des aktuellen Bildes eingelesen wurden. Ab diesem Zeitpunkt muß nicht mehr auf neue Daten gewartet werden, so daß aus diesem Signal das Enable-Signal für alle anderen Module generiert werden kann.

Im Modul `EdgeValueGenerator()` (Figur 27, Tabelle 6) wird die Kanteninformation für jedes einzelne Pixel generiert. Dazu sind jeweils drei Farbwerte nötig (Figur 24 unten), von denen jedoch immer nur zwei neu übernommen werden. Der zentrale Farbwert ist der rechte Farbwert des vorigen Pixels. In den Modulen `EdgeDirUnit()` findet die Entscheidung statt, ob das aktuelle Pixel als Kantenpixel anzusehen ist; im Modul `EdgeDirUnit()` wird der Typ der Kante festgelegt.

Im Modul `EdgeDiffUnit()` (Figur 28) findet die eigentliche Entscheidung statt, ob zwischen zwei Pixeln eine Kante existiert oder nicht. Zu diesem Zweck muß der euklidische Abstand a der beiden Farben im Farbraum bestimmt werden, auf den dann im nachhinein eine Schwellwertbildung angewendet wird. Der Schwellwert

(ref) ist als Parameter ausgeführt worden, damit er leicht geändert werden kann. Grundsätzlich wird ein Schwellwert von 40.0 angenommen, da dieser Wert sich an vielen Testbildern bewährt hat. Der Wert braucht in der endgültigen Hardware-Implementation nicht festgelegt zu werden, sondern kann immer wieder geändert werden, falls sich herausstellt, daß für verschiedene Szenen verschiedene Schwellwerte verwendet werden sollten. In der Regel wird aber bei unbekannte Szenen mit einem Standardwert gearbeitet.

In dem Modul EdgeDirUnit() (Figur 29, Tabelle 8) findet die Bestimmung der Richtung einer Kante statt. Dazu werden die Abstände der einzelnen Farben zum Koordinatenursprung im Farbraum berechnet, die dann untereinander verglichen, die gewünschten Werte liefern.

Hier wird die Berechnung der Formel

$$a = \sqrt{x^2 + y^2 + z^2}$$

approximiert, da die genaue Berechnung der Wurzel viel zu aufwendig für diesen Fall wäre. Als Approximation wird die Formel

$$a_{approx} = k + \frac{1}{4} * l + \frac{1}{4} * m$$

verwendet, wobei $k = \max(x, y, z)$, $l = \text{med}(x, y, z)$ und $m = \min(x, y, z)$ ist. Der maximale Fehler entsteht, wenn $x = y = z$ gilt. In dem Fall liefert die genaue Formel $a = \sqrt{3}x$, während die Approximation $a_{approx} = 3/2x$ ergibt. Der maximale Fehler beträgt demzufolge $F_{max} = (3/2 - \sqrt{3})/3/2 \approx 13,4\%$. Durch eine andere Wahl der Vorfaktoren, $a_{approx} = k + 11/32 * l + m/4$, läßt sich der maximale Fehler sogar auf 8% minimieren. Aufgrund der viel einfacheren Form der Vorfaktoren bei der ersten Variante (keine Multiplikation nötig, sondern nur eine Shift-Operation um zwei Stellen nach rechts) wird der größere Fehler in Kauf genommen. Ein Fehler von 13% mag viel klingen; da die Ergebniswerte aber nur für eine Schwellwertbildung benötigt werden, ist das akzeptabel. Es werden einfach ein paar Pixel weniger als Kante markiert als normalerweise. Durch Anpassen des Schwellwertes läßt sich das jedoch wieder wettmachen.

Das Ergebnis der Berechnung liegt im Fixpunkt-Format (hier 9.2) vor, mit dem dann auch weitergerechnet wird.

Um die Verfolgung der Stufen in den einzelnen Fenstern durchführen zu können, wird die Kanteninformation zeilenweise benötigt. Da die Information aber spaltenweise neu ins Fenster übernommen wird, muß bei dem horizontalen Fenster eine Neuordnung der Bits stattfinden, damit sie in der richtigen Reihenfolge gespeichert werden. Bei dem vertikalen Fenster muß dies nicht geschehen. Dort findet nämlich eine Transformation von vertikal nach horizontal statt. Im Endeffekt wird die Wirkung der Transformation durch die Neuordnung der Bits rückgängig gemacht. In den Blockschaltbildern wurden die Parameter nicht durch konkrete Werte ersetzt, damit das Muster der Verkettung der Bits besser deutlich wird (w ist die Breite des Kantenfenster, also 17, und h ist die Höhe, auch 17).

Bei dem Modul `pipe()` (Figur 35, Tabelle 12) handelt es sich um ein schlichtes Verzögerungsmodul, das ein Datum übernimmt, und nach einer fest vorgegebenen Anzahl von Takten wieder ausgibt. In der Grundform wäre das Modul so zu implementieren, daß nacheinander so viele Register geschaltet werden, wie es die Länge verlangt. Dies ist jedoch aufgrund der beträchtlichen Hardware-Kosten (7 Gatter pro gespeichertes Bit) nicht sinnvoll, da zusätzlich noch die Länge variabel gehalten ist, gar nicht möglich. Es wird daher ein RAM vorgesehen, bei dem die Hardware-Kosten wesentlich günstiger (maximal 1 Gatter pro gespeichertes Bit) sind. Bei dem Entwurf der `Pipe()` wurde mit besonderer Sorgfalt vorgegangen, damit die Hardware-Kosten möglichst gering ausfallen, da die Pipes aufgrund ihrer enormen Größe die höchsten Kosten verursachen. In der normalen Form einer Pipe ist immer ein Dualport-RAM notwendig, da in jedem Takt ein Wert geschrieben und gleichzeitig ein Wert gelesen werden muß. Durch das parallele Ablegen zweier Werte im Speicher konnte der zweite Port eingespart werden, indem immer in einem Takt zwei Werte gleichzeitig an eine Adresse geschrieben werden und im nächsten Takt zwei Werte parallel ausgelesen werden (Figur 37).

Folglich hat das verwendete RAM nun die doppelte Datenbreite, aber nur noch die halbe Länge. Die Speichermenge ist also gleich geblieben, aber es wurde ein Kosten verursachender Port eingespart. Die Register In0 und Out1 werden zum Parallelisieren zweier Daten bzw. Serialisieren der Daten benötigt. Das Register Stat enthält das Schreib-/Lese-Signal für das RAM. Nach den bisherigen Überlegungen wäre es nur möglich, gerade Längen für die Pipe zuzulassen. Durch das zusätzliche Register DIn sind nun auch ungerade Längen möglich, da es für die Verzögerung um einen weiteren Takt sorgt, falls es in den Datenpfad eingeschaltet wird. Der Adreßzähler Adr wird nach jedem Schreibvorgang inkrementiert. Ist die Länge der Pipe erreicht, so wird wieder bei der Adresse 0 begonnen; demzufolge wird das RAM als Ringpuffer betrieben. Um undefinierte Werte im RAM zu vermeiden, wurde zusätzlich ein Reset-Signal verwendet, durch das eine Initialisierungsphase angeworfen wird, die den RAM-Inhalt löscht. Pro Takt wird immer eine neue Zelle gelöscht; also sind bei einer Länge n des RAMs dementsprechend n Takte erforderlich. Während dieser Zeit werden die Daten am Eingang DIn ignoriert.

In dem Modul MaskGenerator(), Figur 38, Tabelle 13 werden die beiden Masken generiert, mit denen das aktuelle Kantenfenster maskiert wird, um nur relevante Pixel aus der Umgebung zu betrachten. Die Notwendigkeit der y-Maske entsteht dadurch, daß bei der Bearbeitung der Pixel der ersten 8 Zeilen im oberen Bereich des Fensters noch Pixel enthalten sind, die gar nicht zum aktuellen Bild gehören. Um zu verhindern, daß unter Umständen über die Bildgrenze hinaus Geraden verfolgt werden, erfolgt eine Maskierung dieser Pixel. Entsprechendes gilt für die untersten 8 Zeilen des Bildes, bei denen in den unteren Zeilen keine legalen Kantenwerte vorhanden sind.

Das Kantenfenster wird, wie schon erwähnt, bei der Bearbeitung laufend um ein Pixel nach rechts weitergeschoben. Ist das letzte Pixel einer Bildzeile erreicht, so wird ein Sprung zum ersten Pixel der nächsten Zeile ausgeführt. Durch dieses Vorgehen entstehen bei den letzten acht Pixeln einer Zeile am rechten Rand des Fensters Pixel, die eigentlich schon zum linken Rand des Bildes gehören (vergl. Figur 39). Entsprechend sind im Fenster bei den ersten acht Pixeln einer Zeile noch Werte enthalten, die eigentlich zum rechten Rand des Bildes gehören. Um die jeweils für das aktuelle Fenster ungültigen Werte auszublenden, wird die x-Maske benötigt. Dadurch, daß im nicht maskierten Fenster sowohl Pixel vom rechten, als auch vom linken Rand des Bildes enthalten sein können, geht beim Umschalten in die nächste Zeile keine Zeit verloren, und es kann eine Behandlung wie im Normalfall stattfinden.

In dem Modul MaskCounter() (Figur 40, Tabelle 14) wird die Position des aktuell zu bearbeitenden Pixels bestimmt. Zu Beginn stehen die Zähler außerhalb des Bildes $(x,y) = (-11,-9)$, da über die ersten Takte zunächst einmal die Kanteninformation für die ersten Zeilen generiert werden muß; dementsprechend gibt das Signal solange ein Valid = 0 aus. Die Variablen x_in und y_in bestimmen, ob das neu im Kantenfenster sichtbar werdende Pixel zur Umgebung des aktuellen Pixels gehört. Das Signal EndOfLine wird jeweils am Ende einer Zeile aktiv, um das Toggeln der xmask im Modul MaskGenerator() zu ermöglichen.

Für die Verfolgung der Geraden stehen bisher zwei Kantenfenster zur Verfügung, von denen im Normalfall nur eines im Zentralpixel als Kante markiert ist. Durch die UND-Verknüpfung des gesamten Fensters mit diesem Zentralpixel wird das nicht benötigte Kantenfenster ausgeblendet. In einigen Fällen können jedoch auch beide Fenster erhalten bleiben, so daß die beiden Fenster pixelweise ODER-verknüpft werden.

Das Untermodul CutNeededWindow() hat damit keine eigenständige Funktionalität.

In ihm werden nur aus dem 289 Bit breitem Wire die richtigen Bits ausgewählt, was einer einfachen Verdrahtung entspricht, und somit auch keine Zeit kostet.

In MaskNeededWindow() erfolgt dann die eigentliche Maskierung des nun nicht mehr rechteckigen Fensters, bei dem die Pixel ausgeblendet werden, die nicht zur aktuellen Umgebung gehören.

Im Modul GetPosInEdge() (Figur 44, Tabelle 17) wird aus dem Kantenfenster der Anfangs- und Endpunkt der verfolgten Geraden extrahiert. Im Modul FindEnd() (Tabelle 18) wird die zentrale Stufe bis zu ihrem Ende in beide Richtungen verfolgt. An den Enden ergeben sich die Stati, die angeben, in welchen Richtungen weitere mögliche Stufen liegen. Durch GetBestStat() (Figur 45, Tabelle 19) werden die bestmöglichen Stati ausgeblendet. Im Modul SortSections() wird das Kantenfenster in vier Sektoren unterteilt (wieder kein richtiges Modul, sondern nur Auswahl der Bits), worauf dann anhand der bestmöglichen Stati nach rechts und links jeweils der Sektor ausgewählt wird, in dem weitere Stufen vermutet werden. In findSeq() (Tabelle 22) werden alle möglichen weiteren Stufen im entsprechenden Sektor verfolgt. SelectJumps() (Tabelle 23, Figur 47) dient schließlich dazu, aus allen möglichen Stufen nur die Stufen auszuwählen, durch die wirklich eine Gerade interpoliert werden kann. Aufgrund der Stati wird dann das Vorzeichen der y-Werte angepaßt.

Im Modul findEnd() wird die zentrale Stufe bis zu ihrem Ende verfolgt. Prinzipiell ist dies ein iterativer Prozeß, denn an jeder Stelle (ausgehend von der Mitte) erfolgt ein Vergleich, ob die Stufe schon beendet ist. Ist das der Fall, so wird der Status an dieser Stelle bestimmt, andernfalls erfolgt an der nächsten Stelle die gleiche Prozedur. Da dies jedoch (diskret aufgebaut) eine zu lange Zeit in Anspruch nehmen würde, wurde dieses Modul als einzelnes "Case" modelliert, bei dem die Abfrage parallel stattfindet. Durch eine automatische Logik-Optimierung mittels eines geeigneten Tools wurden so die Hardware-Kosten, und auch die Bearbeitungszeit reduziert.

Am Ende einer Stufe kann es vorkommen, daß sich sowohl nach oben, als auch nach unten eine weitere Stufe zur Verfolgung anbietet. Anhand des jeweils entgegengesetzten Status am Ende der Stufe wird die beste Möglichkeit ausgewählt. Bietet sich keine Stufe an, so wird der entsprechende Status auf NO korrigiert.

Da die letzten Stufen nicht mehr so lang sein können, wird deren Bitbreite auf zwei bzw. ein Bit reduziert. (Sie können nicht mehr so lang ausfallen, weil, wenn bei der fünften Stufe noch etwas eingetragen werden soll, die vorigen Stufen mindestens die Länge eins besessen haben müssen. Folglich kann die fünfte Stufe noch maximal die Länge drei haben, damit sie noch vollständig im Fenster liegt und erkannt wird.)

Da dieses Modul wieder zu komplex für einen diskreten Aufbau ist, wurde wieder auf die Möglichkeit der Modellierung mittels eines Case zurückgegriffen, und mittels einer Logik-Optimierung minimiert.

Im Modul GetPossibleJumps (Figur 48) werden die Längen der einzelnen Stufen mit der der zentralen Stufe verglichen, da sich nur eine Gerade ergeben kann, wenn alle

Für die Bestimmung der Geraden wird der Kehrwert des x-Abstandes der beiden

Bei diesem Modul und dem anschließenden MaskMixFactors() ist eine Besonderheit anzumerken. Die beiden Module arbeiten nämlich überlappend. Während das Modul ComputeMixFactor() noch damit beschäftigt ist, den Mischfaktor zu berechnen, sind die Länge len und das zusätzliche Flag no_good bereits an MaskMixFactors() weitergeleitet worden, und befinden sich dort in Bearbeitung. Der Ausgang mix ist also immer um ein Takt verzögert, so daß insgesamt ein Bearbeitungstakt und damit zusätzliche Register eingespart werden konnten.

Für die Berechnung des linken Zumischfaktors, wird die verfolgte Gerade des vorherigen Pixels bezüglich des rechten Mischfators wiederverwendet, da die beiden Geraden vollkommen identisch sind. Lediglich im Fall des ersten Pixels in einer Zeile gilt dies nicht, so daß über xmask der entsprechende Mischfaktor und auch alle zusätzlichen Informationen gelöscht werden, Modul CompuLeftEdge, Figur 53, Tabelle 29).

Die bisher berechneten Mischfaktoren wurden immer nur aufgrund einer einzigen Geraden bestimmt. Um zu einem globaleren Eindruck zu gelangen und somit die bestmöglichen Mischfaktoren für ein Pixel zu bestimmen, werden im Modul `MaskMixFactor()` alle vier Geraden betrachtet und, falls nötig, werden einige Mischfaktoren noch ausmaskiert. Die Notwendigkeit entsteht durch Rauschen, welches durch die Verwendung des Differenzoperators als Kantenerkennung resultiert (auch jeder andere Kantenoperator würde in irgendeiner Weise Rauschen produzieren). Als Beispiel betrachte man eine horizontal orientierte Kante zwischen zwei Farben. Im horizontalen Kantenbild ergibt sich das gewohnte Bild der erwünschten Sprünge. Im vertikalen Kantenbild wird jedoch bei jedem Sprung auch ein Kantenpixel markiert, welches aber keine sinnvolle Bedeutung hat. Da bei der Verfolgung einer Geraden dieser Fall nicht erkannt wird, werden diese Fälle unter

Die tatsächliche Farbe des Pixels wird aufgrund seiner ursprünglichen Farbe und der Farben der vier Nachbarn bestimmt. Zu diesem Zwecke werden die benötigten Farben geladen, wobei nacheinander die rechte Farbe als zentrale Farbe des nächsten Pixels und linke Farbe des übernächsten Pixels fungiert. In `ComputeCenterFactor()` wird der noch fehlende Mischfaktor für das zentrale Pixel bestimmt, worauf dann in `Mixing_channel()` des Moduls `Mixing()` (Figur 56, Tabelle 31) die eigentliche Farbmischung pro Farbkanal stattfindet. ✓

Die endgültige Mischung der Farben findet nach der Mischungsformel im Modul `Mixing_Channel()` (Figur 58, Tabelle 33) statt. Nach der unter Umständen nötigen Division durch zwei, findet eine Rundung statt, mit der die Genauigkeit noch etwas erhöht werden kann, worauf sich ein Abfangen eines Überlaufes anschließt.

Figur 59 enthält links eine mögliche Systemkonfiguration: Der Renderer schickt die

Farbwerte der rasterisierten Pixel und weitere Informationen (x,y-Position, z-Wert für z-Buffering) an beide Framebuffer, von denen jedoch nur einer die Werte in den Speicher übernimmt. Der Inhalt des anderen Framebuffers wird an den RAMDAC übertragen. Um eine möglichst hohe Bildwiederholrate zu gewährleisten, d.h., um Bildflimmern zu vermeiden, wird meistens nicht nur ein Pixel pro Takt an die RAMDAC übertragen, sondern zwei oder auch vier; so kann die Takt-rate außerhalb des Chips klein gehalten werden, und trotzdem eine hohe Übertragungsrate erreicht werden. Die Pixel werden in der RAMDAC dann wieder serialisiert und mit einer entsprechend vielfachen Taktrate an den Monitor gesendet. Die Idee, das Post-Antialiasing im Display-Prozeß einzubauen, beruht darauf, daß einfach nur der normale RAMDAC gegen einen um die Antialiasing-Funktion erweiterte RAMDAC ausgetauscht werden kann (Figur 60). Der Renderer und der Framebuffer sind von dieser Änderung nicht betroffen, so daß es möglich wird, Antialiasing auch in den Systemen einzubauen, die gar nicht dafür vorgesehen worden sind. Dies ist die Besonderheit des vorliegenden Verfahrens, die von den bisherigen Antialiasing-Verfahren nicht geboten worden ist.

Ein Beispiel eines RAMDAC befindet sich in Figur 61. In diesem speziellen Fall können bis zu vier Pixel pro Takt übernommen werden. Da das Anti-aliasing-Verfahren aber in der Grundvariante immer nur ein Pixel pro Takt bearbeiten kann, kann es nicht einfach vor dem RAMDAC geschaltet werden. Für den Einbau sind zwei Alternativen günstig:

1. Einbau vor dem RAMDAC

Hier müssen immer vier Pixel gleichzeitig pro Takt bearbeitet werden; die gesamte Logik müßte also vier mal parallel aufgebaut werden. Der Speicheraufwand ändert sich nicht. In jedem Takt wird das Kantenfenster um vier Pixel weitergeschoben, wobei jede Logikeinheit immer einen der vier Pixel bearbeitet.

2. Einbau in dem RAMDAC nach dem Demultiplexer

Nach dem Demultiplexer muß immer nur ein Pixel pro Takt bearbeitet werden, da durch das Latch und den anschließenden Demultiplexer die vier parallel vorliegenden Pixel serialisiert werden, jedoch wird hier eine höhere Taktrate benötigt. Die Taktrate in diesem Bereich ist nicht konstant, sondern richtet sich nach der Bildschirmauflösung und der erwünschten Bildwiederholrate. Der

Das zuvor beschriebene Verfahren des Double-Buffers kann auf einen Triple-Buffer erweitert werden, wobei der Inhalt eines Framebuffers auf dem Monitor dargestellt wird, der zweite mittels des Post-Antialiasings bearbeitet wird, und im dritten Framebuffer wird bereits das nächste Bild durch den Renderer erzeugt. Dieser Vorgang ist in Figur 62 schematisch als Zyklus angedeutet. Als Blockschaltbild ist ein Tripelpuffer in Figur 63 wiedergegeben. Drei Framebuffer sind parallelgeschaltet und erhalten ihre Signale zeitlich versetzt in zyklischer Vertauschung bzw. geben sie entsprechend ab.

Der Ablauf bei Verwendung eines Triple-Buffers wird nun wie folgt beschrieben:

Ist das Antialiasing eines Bildes beendet, so wartet der Prozeß auf die Fertigstellung des nächsten Bildes durch den Renderer. Steht das nächste Bild zur Verfügung, so übernimmt der bisherige Renderer-Framebuffer die Funktion des Antialiasing.

Ist der Rendererprozeß mit der Generierung eines neuen Bildes fertig, so wird

unverzöglich auf den ursprünglichen Display-Buffer umgeschaltet, wo sofort mit der Bearbeitung eines neuen Bildes begonnen werden kann. Der Rendererprozeß muß keinesfalls auf den Displayprozeß warten, da aufgrund der schon beschriebenen Zeitverhältnisse das Anzeigen des Bildes an den Antialiasing-Buffer übertragen wurde.

Für die Verwendung im Triple-Buffer-Betrieb müssen einige Änderungen am Grundsystem des Antialiasing vorgenommen werden, wie es in Figur 64 dargestellt ist.

Für den Triple-Buffer-Prozeß ist es nicht notwendig, die Farbwerte nach der Kantengenerierung aufzuheben, da sie noch einmal aus dem Speicher gelesen werden können. Beim Display-Prozeß bestand diese Option nicht, da jeder Farbwert nur einmal übertragen wurde. Demzufolge werden die Farbwerte, die aus dem EdgeValueGenerator() kommen, verworfen (wie man an dem nicht beschalteten Ausgang sieht) und erst dann wieder aus dem Speicher geholt, wenn sie für die Mischung der Farben benötigt werden.

Die Speicherzugriffe verdoppeln sich nicht, wie eigentlich anzunehmen wäre, denn beim zweiten Auslesen müssen nicht alle Farbwerte noch einmal gelesen werden, sondern nur diejenigen, die wirklich zur Farbmischung benötigt werden. Statistische Untersuchungen ergaben, daß im schlimmsten Fall 20% der Pixel des Bildes noch einmal gelesen werden müssen. Natürlich müssen auch nicht alle Pixel in den Speicher zurückgeschrieben werden, sondern nur diejenigen, die wirklich eine Farbveränderung erfahren haben (erkenntlich durch das Ausgangssignal mixing des Moduls Mixing()), was in maximal 10% der Fälle geschieht. Statistisch gesehen müssen pro Pixel demnach 1,3 Speicherzugriffe geschehen (100% Bild einmal auslesen, 20% zu mischende Pixel noch einmal lesen, 10% der Pixel zurückschreiben).

Die kombinatorische Logik des Antialiasing-Systems ließe es an sich zu, daß in jedem Takt ein weiteres Pixel fertiggestellt wird, dies ist jedoch entbehrlich. Um bei einer Bildschirmauflösung von 1280 x 1024 die geforderte Bildgenerierungsrate von 25 Bildern pro Sekunde zu erreichen, müssen pro Sekunde $1280 \times 1024 \times 25 = 32,768$ Millionen Pixel bearbeitet werden. Bei einer angenommenen Taktrate von 66MHz für das System stehen also pro zu bearbeitendes Pixel 2,01 Takte Zeit zur Verfügung. Bei höheren Auflösungen kann dieser Wert noch kleiner werden, jedoch wird dann der Renderer keine Bildgenerierungsrate von 25 Bilder pro Sekunde

gewährleisten können. Bei heute üblichen Speichern kann man nicht davon ausgehen, daß in jedem Takt ein Datum übertragen werden kann, da Refresh-Zeiten für die Erhaltung der Speicherinhalte bei dynamischen RAMs und Rowwechselzeiten eingehalten werden müssen. Die effektive Übertragungsrate hängt stark von der Zugriffsreihenfolge auf den Speicher ab, so daß allgemein keine konkrete Übertragungsrate angegeben werden kann.

Die Speicherschnittstelle ist in der Regel jedoch so dimensioniert, daß nicht nur ein Pixel pro Takt aus dem Speicher übertragen wird, sondern parallel gleich mehrere. So können parallel bis zu 4 Pixel übertragen werden.

Die wesentliche Voraussetzung, um solche Übertragungsraten zu gewährleisten, ist eine effiziente Speicherabbildung, so daß möglichst schnell auf die nächsten benötigten Pixel zugegriffen werden kann. Als Beispiel sei hier die Speicherorganisation im Framebuffer eines bekannten Systems (GMD VISA) erwähnt. Der verwendete SDRAM-Speicher (1Mx32-Organisation) ist physikalisch in zwei Bänke aufgeteilt, die aus Rows (2048) aufgebaut sind, die wiederum in Columns unterteilt sind (256 / in einer Column ist jeweils ein Pixel abgelegt). Innerhalb einer Row ist es möglich, auf jede beliebige Columnadresse in einem Takt zuzugreifen, wohingegen das Umschalten der einzelnen Rows 7 Takte in Anspruch nimmt. Die Speicherabbildung wurde also so ausgelegt, daß die Anzahl der Row-Wechsel sowohl beim Renderingprozeß, als auch beim Auslesen für den Display-Prozeß minimiert wird. Um einen weiteren Adreßbus zu sparen, werden mit einer Adresse immer 2 benachbarte Pixel gleichzeitig aus dem Speicher geholt, da 2 Speicherchips zur Verfügung stehen. Die Rows wurden nun so unterteilt, daß in ihnen ein Bereich von 32 x 16 Doppelpixeln Platz findet, und somit innerhalb dieses Bereichs jedes Doppelpixel in einem Takt auslesbar ist (Figur 65).

Die verwendete Speicherabbildung kommt dem hier verwendeten Verfahren auch sehr entgegen. Beim Auslesen der Pixel für die Kantengenerierung wird das komplette Bild linear zeilenweise ausgelesen, d.h., es ist möglich, in 32 Takten 64 Pixel aus dem Speicher zu lesen, ehe ein Row-Wechsel erfolgen muß, der 7 Takte benötigt. Für diesen Prozeß ergibt sich demnach eine Übertragungsrate von 64 Pixeln in 39 Takten = 1,64 Pixel/Takt. Jedoch ist dies nicht der einzige Prozeß der auf dem Speicher abläuft. Es werden konkurrierend dazu die schon angesprochenen 20% der Pixel für die Mischung geholt, und 10% der Pixel wieder zurückgeschrieben. Durch geeignetes Auspuffern der Anforderungen ist es möglich, die Rowwechsel dazu zu nutzen, um zwischen den einzelnen Prozessen umzuschalten.

Für den zweiten Auslese-Prozeß und den Schreibe-Prozeß sollten die Anforderungen so lange gesammelt werden, bis alle eine Row betreffende Anfragen auf einmal bearbeitet werden können, so daß wiederum möglichst wenige Row-Wechsel sogar bei diesen nicht mehr linearen Vorgängen durchgeführt werden müssen, wie es in Figur 66 dargestellt ist.

Der Ablauf im Modul RAM-Schnittstelle() für die aktuell aus dem Speicher zu holenden Pixel ist wie folgt ausgebildet:

- Lineares Auslesen aus dem Speicher die Pixel, die für die Kantengenerierung benötigt werden, und schreibe sie in die EdgeReadFIFO() bis du am Ende einer Row angekommen bist. Hierfür wird intern ein Zähler verwendet, der sich merkt, bis zu welchem Pixel gelesen wurde.
- Am Ende einer Row wird kontrolliert, wie der Status der anderen beiden Prozesse jeweils aussieht. Sind in einer der beiden FIFOs (RequestFIFO() und WriteFIFO()) alle eine Row betreffende Anfragen eingegangen, so werden sie hintereinander an den Speicher weitergeleitet. Ist dies nicht der Fall, wird weiter die EdgeReadFIFO() gefüllt.

Die EdgeReadFIFO() ist also dimensioniert, daß sie möglichst nie leer wird, so daß immer weiter gearbeitet werden kann.

Die aus MaskMixFactor() herauskommenden Mischfaktoren werden mittels des Moduls cmp() darauf hin überprüft, welcher der Mischfaktoren ungleich Null ist. Sollte dies bei keinem der Fall sein, so muß das aktuelle Pixel nicht gemischt werden, und die Mischfaktoren werden verworfen. Ist jedoch mindestens einer der Faktoren ungleich Null, so werden sie zusammen mit der aktuellen Pixelnummer in der MixFactorFIFO() abgespeichert, um sie dann später (sobald die Farbwerte aus dem Speicher geholt wurden) für die eigentliche Mischung zu verwenden. Ausgehend von den Mischfaktoren werden dann in der RequestFIFO() die benötigten Farbanforderungen zwischengespeichert. Bei benachbarten Pixeln werden unter Umständen die gleichen Farbwerte benötigt, so daß diese doppelten Anforderungen aussortiert werden. Sobald alle eine Row betreffenden Anfragen eingegangen sind, können sie an die RAM-Schnittstelle() übermittelt werden, die dann daraufhin die Farbinformation an die ReturnFIFO() zurückgibt. In der ReturnFIFO() werden dann aufgrund der am Ausgang der MixFactorFIFO() stehenden Mischfaktoren die benötigten Farbinformationen parallelisiert, und sobald alle benötigten Farben

Die Erfindung beschränkt sich in ihrer Ausführung nicht auf die vorstehend angegebenen bevorzugten Ausführungsbeispiele. Vielmehr ist eine Anzahl von Varianten möglich, welche von der dargestellten Lösung auch bei grundsätzlich anders gearteten Ausführungen Gebrauch macht.

Patentansprüche

1. Verfahren zum Eliminieren unerwünschter Stufungen an Kanten bei Bilddarstellungen im Zeilenraster, insbesondere im On-line Betrieb, gekennzeichnet durch die Schritte:
 - a) Anwendung eines Kantenoperators auf einen Bildteil zur Grobermittlung mindestens eines gerasterten Kantenverlaufs,
 - b) Bestimmung der Position mindestens eines ersten Pixels aus der Menge derjenigen Pixel, die den gerasterten Kantenverlauf bilden oder an diesem gerasterten Kantenverlauf angrenzen,
 - c) Approximation einer Geraden zur Ermittlung eines wahrscheinlichen Verlaufs der ungerasterten Bildkante in der Nähe des ersten Pixels,
 - d) Ermittlung eines Kriteriums aus der Approximationsgeraden und der Position des ersten Pixels für die Zumischung einer Farbe X zu der Farbe C im betrachteten ersten Pixels und
 - e) Mischung der ermittelten Farbe X zu der Farbe C im betrachteten ersten Pixel.
2. Verfahren nach Anspruch 1, dadurch gekennzeichnet, daß das Kriterium des Verfahrensschrittes d) in Abhängigkeit von der Lage des betrachteten Pixels relativ zur Approximationsgeraden festlegt, welche Farbe X zu der Farbe C des betrachteten Pixels zugemischt wird.
3. Verfahren nach Anspruch 2, dadurch gekennzeichnet, daß das Kriterium gemäß Verfahrensschritt d) in Abhängigkeit von der Lage des betrachteten Pixels relativ zur Approximationsgeraden festlegt, daß die Farbe mindestens eines benachbarten Pixels gewichtet zur Farbe des betrachteten Pixels zugemischt wird.
4. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet,

zeichnet, daß bei einem betrachteten Pixel, das von der Approximationsgeraden nicht geschnitten wird, die Farbe unverändert bleibt.

5. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß bei einem betrachteten Pixel, das von der Approximationsgeraden geschnitten wird, die resultierende Farbe R nach Maßgabe des folgenden Kriteriums bestimmt wird:

die Approximationsgerade teilt das betrachtete Pixel in zwei Teilflächen F_1 , F_2 , wobei $F_1 + F_2 = 1$, mit 1 ist die Gesamtfläche des Pixels, wobei F_1 diejenige Teilfläche ist, in welcher der Pixelmittelpunkt liegt;

- zugemischt wird zu der Farbe C des betrachteten Pixels die Farbe X desjenigen benachbarten Pixels, welche an die längste vom Raster gebildete Kante der Teilfläche F_2 angrenzt.

6. Verfahren nach Anspruch 5, dadurch gekennzeichnet, daß sich die resultierende Farbe R aus der ursprünglichen Farbe C des betrachteten Pixels und der zugemischten Farbe X eines benachbarten Pixels nach folgender Gleichung ergibt:

$$R = F_1 \times C + F_2 \times X$$

7. Verfahren nach Anspruch 5 oder 6, dadurch gekennzeichnet, daß die Teilflächen F_1 , F_2 durch ein geeignetes Approximationsverfahren approximiert werden.
8. Verfahren nach einem der vorstehenden Ansprüche, dadurch gekennzeichnet, daß die genannten Verfahrensschritte auf einen mittels eine Rendering und/oder Shading-Verfahrens behandelten Bildteil angewendet werden.
9. Verfahren nach Anspruch 8, dadurch gekennzeichnet, daß das Shading-/Rendering dreiecks- oder Scanlinebasiert ist, oder daß es sich um ein Gouraud-oder Phong-Shading handelt.
10. Verfahren nach einem der vorstehenden Ansprüche, dadurch gekennzeichnet, daß die vorgenannten Verfahrensschritte a) bis e) einzeln oder in

Gruppen im zeitlichen Versatz ausgeführt werden.

11. Verfahren nach Anspruch 10, dadurch gekennzeichnet, daß der zeitliche Versatz mindestens eine Bildzeile beträgt.
12. Verfahren nach einem der vorstehenden Ansprüche, dadurch gekennzeichnet, daß die Verarbeitung im zeitlichen Versatz in einem Framebuffer ohne weitere Zwischenspeicherung erfolgt.
13. Verfahren nach einem der vorstehenden Ansprüche, dadurch gekennzeichnet, daß die Approximationsgerade über mehrere Stufen des gerasterten Kantenverlaufs verläuft, und daß die Approximationsgerade endet, wenn die Kriterien
 - 1) Es können maximal zwei verschiedene Stufenlängen vorkommen, deren Stufenlängen sich außerdem um maximal 1 unterscheiden dürfen.
 - 2) Nur eine der beiden Stufenlängen darf mehrmals hintereinander auftreten.
 - 3) Durch das Aneinanderreihen der Anzahlen der Stufen, die die gleiche Länge haben, erhält man eine Zahlensequenz, bei der abwechselnd immer eine Eins und dann eine beliebige Zahl (> 0) steht. Die Einsen (nur die an jeder zweiten Position) werden aus dieser Sequenz gestrichen. Bei der erhaltenen Sequenz dürfen wieder nur zwei verschiedene Zahlen vorkommen, die sich um eins unterscheiden.
 - 4) Bei der unter 3. erhaltenen Sequenz darf nur eine der beiden möglichen Zahlen mehrmals hintereinander auftreten.
 - 5) Durch wiederholtes Anwenden der Regeln 3. und 4. auf die Zahlensequenz, läßt sich ein immer globalerer Blick auf die Kante gewinnen.

in aufsteigender Reihenfolge überprüft werden und mindestens ein Kriterium nicht erfüllt ist.

14. Verfahren nach einem der Ansprüche 1 bis 13, dadurch gekennzeichnet, daß die Approximationsgerade über mehrere Stufen des gerasterten Kantenverlaufs verläuft, und daß die Approximationsgerade endet, wenn eines der Kriterien
- 1) Es können maximal zwei verschiedene Stufenlängen vorkommen, deren Stufenlängen sich außerdem um maximal 1 unterscheiden dürfen.
 - 2) Nur eine der beiden Stufenlängen darf mehrmals hintereinander auftreten.
 - 3) Durch das Aneinanderreihen der Anzahlen der Stufen, die die gleiche Länge haben, erhält man eine Zahlensequenz, bei der abwechselnd immer eine Eins und dann eine beliebige Zahl (> 0) steht. Die Einsen (nur die an jeder zweiten Position) werden aus dieser Sequenz gestrichen. Bei der erhaltenen Sequenz dürfen wieder nur zwei verschiedene Zahlen vorkommen, die sich um eins unterscheiden.
 - 4) Bei der unter 3. erhaltenen Sequenz darf nur eine der beiden möglichen Zahlen mehrmals hintereinander auftreten.
- oder eines der Kriterien 1), 2), 3) oder eines der Kriterien 1), 2) nicht erfüllt ist.
15. Verfahren nach einem der der vorhergehenden Ansprüche, gekennzeichnet durch das Vorsehen eines Tripple-Buffers, wobei sich die drei resultierenden Buffer in zyklischer Vertauschung parallel die Verfahrensschritte Rendering, Post-Antialyasing und Bildwiedergabe teilen.

SECRET

—

-

—